KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

VERSLO INFORMATIKOS KATEDRA

Eglė Blažaitytė

Sistemų su klaidų įterpimu formalizavimas

Magistro darbas

Darbo vadovas

Prof. habil. dr. H. Pranevičius

Kaunas, 2007

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

VERSLO INFORMATIKOS KATEDRA

Eglė Blažaitytė

Sistemų su klaidų įterpimu formalizavimas

Magistro darbas

Recenzentas

Prof. habil. dr. A. Avižienis

2007-05-25

Vadovas

Prof. habil. dr. H. Pranevičius

2007-05-25

Atliko

IFM-1/1 gr. stud.

Eglė Blažaitytė

2007-05-20

Kaunas, 2007

TURINYS

ĮVADAS

Daugumai šiuo metu kuriamų realaus pasaulio sistemų yra keliami labai aukšti reikalavimai – saugumas, gyvybingumas, greita reakcija į tam tikrus sistemą veikiančius įvykius, patikimumas. Tokios sistemos realiame gyvenime mus supa visur – nuo paprastos buitinės technikos iki kritinių sistemų. Norint, kad sistema atitiktų jai keliamus reikalavimus, labai svarbu jos kūrimo pradžioje susidaryti aiškų sistemos veikimo vaizdą bei specifikaciją. Realiame pasaulyje neegzistuoja nei viena sistema, kuriai nekiltų sutrikimų - klaidos ar neteisingo veikimo, galimybė. Tokius sutrikimus dažniausiai sąlygoja įvairūs aspektai – žmogiškasis faktorius, sistemos kūrimo metu paliktos arba neapgalvotos klaidos ir t.t.

Kritinės sistemos privalo būti ypatingai patikimos net ir atsiradus klaidai sistemoje, tos sistemos veikimo metu. Sistema turi veikti taip, kaip yra numatyta reikalavimuose, net ir klaidos atveju, nedarydama žalos aplinkai. Dažniausiai kuo didesnis yra sistemos naudingumas, tuo didesni tokios sistemos sutrikimo padariniai. Sistemoje atsiradusių klaidų pasėkoje sistema visiškai nustoja teikti arba teikia neteisingai numatytas paslaugas ar rezultatus, o tai sąlygoja finansinius nuostolius, kartais gali netgi kelti pavojų žmonių, naudojančių tokias sistemas, gyvybei. Sistemoje atsiradę sutrikimai gali būti šalinami sistemos funkcionavimo metu, tačiau yra ir galimybė tai numatyti iš anksto, dar projektavimo metu, leidžiant pačiai sistemai susidoroti su klaida. Prieš kuriant tokią sistemą reikia aiškiai nustatyti aiškius sistemos reikalavimus: kaip sistema turi veikti, ko iš jos tikimasi, kokio lygio turi būti jos patikimumas, saugumas ir kitos esminės savybės. Iš anksto įvertinant galimas klaidas, kurios gali, tačiau nebūtinai privalo, įvykti, galima sukurti klaidoms tolerantišką sistemą. Tikslus tokios sistemos aprašymas yra pradinis ir esminis tolerantiškos klaidoms sistemos kūrimo aspektas.

Neegzistuoja nei viena sistema be tikimybės, kad tos sistemos veikimo atveju įvyks klaida ir darbas bus sutrikdytas. Tokios sistemos teorinis idealaus funkcionavimo modelis parodo, kaip sistema veiks idealiu atveju – laikantis tam tikrų prielaidų ji atliks nustatytus veiksmus bei duos laukiamus ir patikimus rezultatus. Tokių modelių aprašymui yra labai patogu naudoti įvairius formalizmus, kuriuos vėliau galima naudoti imitaciniam modeliavimui.

Tolerantiškų klaidoms sistemų kūrimas yra pakankamai brangi sritis, reikalaujanti nemažai prototipų, panašių sistemų analizės tam, kad nuspręsti ar tikrai tokios sistemos kūrimas atsipirks.

Problemos aktualumas. Kiekvienos sistemos kūrimo tikslas yra veikianti, gyvybinga ir saugi sistema, teikianti norimus ir patikimus rezultatus. Sistemos saugumas – tai sistemos savybė, reiškianti, kad sistemos funkcionavimo metu neįvyks jokia nenumatyta situacija. Gyvybingumas – sistemos reakcija į tam tikrus įvykius ir sugebėjimas atlikti nustatytas užduotis bei pateikti teisingus sprendimus arba rezultatus. Norint sukurti tokią sistemą, kuri ateityje tenkins nustatytus reikalavimus, yra labai svarbu iš anksto nustatyti jos formalią reikalavimų specifikaciją, nes nuo to priklauso galutinis produktas – kiek įvairių situacijų, į kurias sistema gali patekti, ar bus numatyta, kaip ji susidoros su atitinkamais išoriniais ar vidiniais įvykiais. Tokią specifikaciją galima praplėsti įvairiomis modifikacijomis, kurios gali padėti aptikti potencialias klaidas sistemoje, kurias įvertinus kūrimo metu, galima sistemai suteikti tolerancijos klaidoms savybę.

Darbo tikslas. Susipažinti su DEVS ir FDEVS modeliavimo metodika ir pritaikyti ją klaidoms tolerantiškų sistemų modeliavimui, naudojant PLA formalizavimo metodą.

Mokslinis naujumas. FPLA modeliavimo metodikos sukūrimas.

Darbo struktūra. Pirmame šio darbo skyriuje pateikiama klaidos tolerantiškų sistemų savybių, jų kūrimo apžvalga. Taip pat nagrinėjamas teorinis sistemų su įterptomis klaidomis, klaidų hipotezėmis modelis. Antrajame skyriuje apžvelgiami įvairūs sistemos modeliavimo ir formalizavimo metodai, imitacinių modelių rūšys. Trečiajame skyriuje analizuojamas agregatinis formalizavimo metodas, kuriuo tolimesnėse darbo dalyse specifikuojama pasirinkta sistema. Ketvirtojoje darbo dalyje pateikiama DEVS[1] ir FDEVS[2] formalizavimo metodų analizė – struktūra ir pavyzdžiai. Penktajame darbo skyriuje pasirinktoji sistema – alternuojančio bito protokolas – specifikuojamas agregatiniu ir DEVS metodais, nagrinėjant sistemą klaidoms tolerantiškų sistemų aspektu: idealiu funkcionavimo atveju ir su įvestomis klaidomis.

[1] Discrete Event System Specification – diskrečių įvykių sistemos specifikacija

[2] Faulty Discrete Event System Specification - diskrečių įvykių sistemos specifikacija su klaidos hipoteze

1 KLAIDOMS TOLERANTIŠKOS SISTEMOS

Sistemos tolerancija klaidoms – tai savybė, leidžianti sistemai sėkmingai tęsti darbą, esant klaidos įvykiui. Kokybiška sistema sugeba susidoroti su klaida ir toliau teikti jai nustatytas paslaugas bei generuoti laukiamus ir teisingus rezultatus. Jei sistemos veikimo kokybė visiškai sumažėja, šis sumažėjimas yra proporcingas nesėkmės dėl klaidos sunkumui (pvz. neteisingai, neaiškiai kurtoms sistemoms, kuriose net mažiausias sutrikimas gali būti visiško neveikimo priežastis). Tolerancija klaidoms yra ypatingai svarbi savybė visoms aukšto naudingumo, būtino pasiekiamumo ir gyvybiškai svarbioms kritinėms sistemoms. Tai tokios sistemos, kurių net laikinas veikimo sutrikimas gali baigtis mirtimi ar sunkiais sužeidimams žmonėms, žala įrangai ar aplinkai. Tokių sistemų pavyzdžiai - greitosios pagalbos ir įvykių valdymo sistemos, elektros generavimo ciklai, gaisro aliarmo sistema, telekomunikacinių ryšių sistemos ir t.t.

1.1 Klaidoms tolerantiškų sistemų savybės ir kūrimo metodai

Dabartiniai programinės įrangos klaidų toleravimo metodai yra pagrįsti tradicine programinės įrangos tolerancija klaidoms metodika. Šio metodo pagrindinis trūkumas yra tas, kad tradicinė programinės įrangos tolerancija klaidoms buvo sukurta pirmiausia įveikti gamybinėms klaidoms, ir tik po to – aplinkos sukeltoms ir kitoms klaidos įveikti.

Tolerancija klaidoms tai ne tik individualių sistemų savybė, ji taip pat gali charakterizuoti savybes tarp kelių sistemų [1]. Pvz. TCP – perdavimo kontrolės protokolas (*transmission control protocol*) yra sukurtas užtikrinti patikimą dvikryptį komunikavimą paketų perdavimo tinkle, net esant perkrautoms ir nepilnoms komunikavimo sąsajoms. Tai atliekama, informuojant galinius komunikavimo taškus apie galimus paketų praradimus, jų dubliavimąsi, neteisingą eilės tvarką ir iškraipymus taip, kad šios sąlygos nepažeistų duomenų integruojamumo ir jie būtų sėkmingai apdorojami.

Kitas šiek tiek skirtingas klaidos tolerantiškos sistemos pavyzdys- HTML kalba. Ji yra tolimesnio palaikymo (*forward compatible*), o tai reiškia, kad internetinių puslapių naršyklėse gali būti apdorojami ne tik esamos, bet ir vėlesnių HTML kalbos versijų duomenys. Tai leidžia HTML esybėms būti ignoruojamoms tų naršyklių, kurios jų tiesiog nesupranta, tačiau vien dėl to HTML dokumentas nėra neprieinamas – juo galima naudotis be papildomų pastangų.

Galimybė atsigauti po klaidos, klaidoms tolerantiškose sistemose gali būti dalinai charakterizuojama kaip tolimesnis palaikymas (*forward compatible*) ir atgalinis palaikymas (*backward compatible*), t.y. kai sistema nustato, kad įvyko klaida, tolimesnio palaikymo galimybė grąžina sistemos būseną į tą laiką ir klaidą ištaiso, kad ji galėtų veikti toliau. Atgalinio palaikymo savybė grąžina sistemą į tam tikrą ankstesnę būseną, kuri kažkada jau buvo teisinga, pvz. teisingos būsenos nustatymas (*checkpoint*) ir paleidžia ją iš naujo, tačiau pačios klaidos netaiso. Kai kurios sistemos naudoja abi šias savybes tam, kad susitvarkyti su skirtingomis klaidomis ar skirtingomis jų dalimis.

Žvelgiant į individualias sistemas, klaidų tolerancija gali būti pasiekta numatant išskirtines sąlygas kuriant sistemą su hipoteze, kad pat sistema galėtų su tomis išskirtinėmis sąlygomis susidoroti ir save stabilizuoti.

Klaidų toleravimas yra vykdomas tokiais būdais [2]:

1. Klaidos aptikimas – pati sistema turi surasti klaidą;
2. Žalos įvertinimas – reikia surasti tas sistemos dalis, kurias paveikė atsiradusi klaida;
3. Klaidos atstatymas – sistemos klaidingos būsenos atstatymas į teisingą;
4. Klaidos taisymas – sistemos būsenos modifikavimas į tokią, kad būtų užkirstas kelias tolesniam defektų pasikartojimui.

Pagrindiniai klaidų toleravimo metodai yra keli, tačiau dažniausiai išskiriami šie:

1. Gynybinis programavimas.
2. Klaidas toleruojančios architektūros
3. Išimčių (*exceptions*) valdymas.

Gynybinio programavimo metu į kodą įterpiamas perteklinis kodas, kuris tikrina sistemos būseną po tam tikrų modifikacijų tam, kad užtikrinti sistemos teisingumą.

Klaidas toleruojančios architektūros projektuojamos su tikslu, palaikyti programinės ir architektūrinės įrangos perteklingumą bei defektų toleravimo valdiklį, kuris aptinka klaidas bei palaiko sistemos atstatymo galimybę. Dažniausiai tokiose architektūrose egzistuoja keli tokių pačių komponentų moduliai, kurių darbo rezultatai vėliau palyginami.

Išimčių valdymas labai dažnai naudojamas C++ bei Java kalbose – tai galimybė valdyti tam tikras išimtis, prie iš anksto nustatytų sąlygų. Esant tokiam valdymui, sistema gali informuoti vartotoją apie galimas išimtis ir pasirinkti tolesnius veiksmus.

Kuriant klaidoms tolerantiškas sistemas naudojamos įvairios technikos – atstatymo blokai, komponentų dubliavimas, n-versijų programavimas ir t.t.

Pagrindinės tolerantiškų klaidoms sistemų kūrimo technikos [3]:

1. Atstatymo blokai – šis metodas yra paprasčiausias, kai sistema yra išskaidyta į blokus, o tam tikras blokas veikia kartu su arbitru (*adjudicator*), kuris patvirtina įvairių to paties algoritmų įgyvendinimų (implementacijos) rezultatus. Sistemoje su atstatymo blokais, sistemos vaizdas yra padalomas į atstatančius nuo klaidų blokus. Visa sistema yra sukonstruota iš tokių klaidoms tolerantiškų blokų, kurių kiekvienas susideda iš mažiausiai: pirminio, antrinio ir išimtinio kodo, kartu su arbitru. Arbitras yra komponentas, kuris apsprendžia įvairių blokų teisingumą. Prieš nagrinėdamas konkretų vienetą, arbitras pirmiausia įvykdo pirminę alternatyvą (jų gali būti N). Jei arbitras nustato, kad pirminis blokas yra klaidingas, jis bando sugrąžinti sistemos būseną į pradinę ir bando kitą alternatyvą. Jei nėra nei vienos teisingos alternatyvos, arbitras iškviečia išimčių valdiklį (*exception handler*), kuris indikuoja, jog programinei įrangai nepavyko atlikti iškviestos operacijos. Šis metodas padidina poreikį kurti tokias sistemų specifikacijas, kad jos būtų pakankamai išsamios ir turėtų keletą skirtingų alternatyvų, tačiau funkcionaliai būtų tos pačios. Sistema, sudaryta, iš atstatymo blokų, yra komplikuota, nes jai reikalinga galimybė grįžti į pradinę būseną.
2. N- versijų programavimas metodu kiekvienas programinės įrangos sistemos modulis yra padarytas per N skirtingų įgyvendinimo būdų (implementacijų). Kiekvienas variantas atlieka tą pačią užduot ar sistemos operaciją, tačiau skirtingais būdais. Kiekviena versija pateikia savo atsakymą sprendėjui, kuris nustato, ar atsakymas yra teisingas, ir grąžina jį kaip modulio rezultatą. Sistema gali susidoroti su kūrimo klaidomis, tačiau N-versijų programinė įranga faktiškai gali įtraukti keletą techninės įrangos tipų, besinaudojančių keletu versijų programinės įrangos.

   Naudojant N-versijų programinę įrangą, siekiama užtikrinti, kad kiekviena versija būtų įgyvendinta tiek skirtingų būdų, kiek įmanoma, įtraukiant ir skirtingus įrankių rinkinius, ir programavimo kalbas, ir aplinkas. Skirtingos programuotojų grupės, dirbančios prie N-versijų projekto, turi kiek įmanoma mažiau bendradarbiauti tarpusavyje, kad būtų galima išgauti kuo daugiau variacijų.

Skirtumas tarp atstatymo blokų ir N- versijų metodų yra toks, kad atstatymo blokų metodu kiekviena alternatyva turi būti vykdoma pakartotinai iki tol, kol priimtinas sprendimas yra gautas

ir priimtas arbitro, o N-versijų metodas yra sukurtas tam, kad naudoti N-būdų techninę įrangą lygiagrečiai.

Realioms sistemoms pakartotinis alternatyvų vykdymas gali būti per brangus. Kitas skirtumas yra tarp arbitro ir sprendėjo – kiekvienam atstatymo blokui reikalingas arbitras, o N-versijų metode – vienas sprendėjas visai sistemai.

3. Pati save tikrinanti sistema.

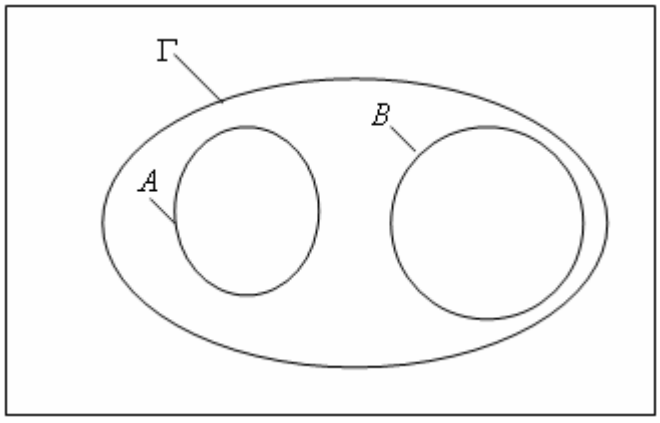
Tai pačios sistemos galimybė tam tikrais momentais kurti atskaitos taškus ir esant poreikiui – į juos grįžti. Šis metodas nėra pakankamai tiksliai aprašytas literatūroje, tačiau dažnai jis būna specifiškai naudojamas svarbiose sistemose – kritinėse saugumo sistemose, pvz. Airbus A-340 lėktuvuose.

1.2 Teorinis sistemų su klaidų įterpimu modelis

Kiekvienos sistemos funkcionavimo metu gali įvykti tam tikra klaida. Egzistuoja įvairūs klaidų klasifikacijos aspektai, tačiau šiame darbe klaidos bus klasifikuojamos pagal:

1. Kilmę – fizinės (vidinės ir išorinės), loginės (dizaino ir sąveikos);
2. Rūšį – praleistos, įverčių ir laikinės;
3. Savybes – trukmės (laikinos ir nuolatinės), sudėties (deterministinės ir nedeterministinės), autonominės (savaiminės ir priklausančios nuo tam tikų įvykių).

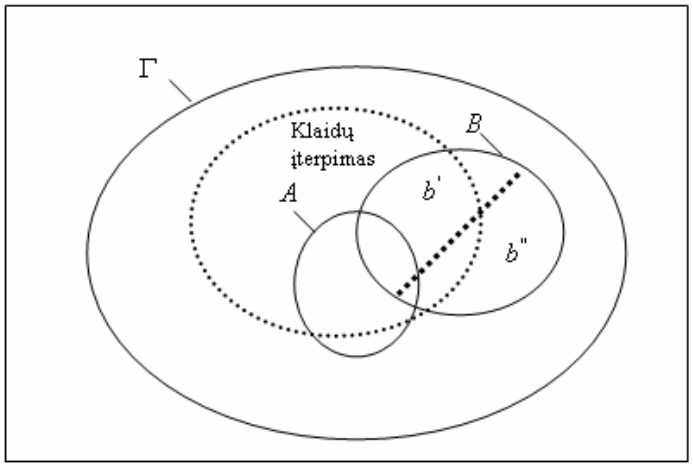
Tarkime, kad yra dvi pagrindinės klasės klaidų, su kuriomis gali susidurti sistema *P* - vidinė ir išorinė [4]. Nors abiejų šių klasių aibės yra itin didelės, yra tik nedidelis kiekis klaidų, kurios turi labai didelę įtaką sistemai - tai tos klaidos, kurios įtakos sistemą po jos įdiegimo [4]. Tarkime, kad turime sistemą *P* ir analizės metu sistemai *P* nebus atlikta jokių modifikacijų.



1 pav. Klaidų klasės, kur A yra vidinių klaidų aibė, o B - išorinių klaidų aibė

1 paveikslėlyje Γ yra visų nepageidaujamų klaidų, kurios gali įtakoti sistemos *P* išvedamą informaciją per visą jos gyvenimo ciklą, aibė. Sąlyga, kad *visi* aibės Γ nariai įtakos *P* išvedamą informaciją, nėra būtina. Vidinių klaidų aibė *A* pažymi, kad dalis Γ aibės yra sukelti sistemos *P* loginių klaidų, o išorinių klaidų aibė *B* pažymi, kad šios klaidos yra sukeltos žmogiškojo faktoriaus arba išorinių techninės ir programinės įrangos klaidų, su kuriomis gali būti susidurta sistemos *P* veikimo metu.

Dalis Γ narių yra nežinomi ir Γ aibė gali būti prilyginta juodajai dėžei. Aibė *A* gali reikšti absoliučiai begalinę ir nežinomą klaidų aibę arba tuščią klaidų aibę. *B* yra absoliučiai begalinė ir dalinai nežinoma klaidų aibė, $A \cup B \approx \Gamma$. Aibėje Γ esantys nariai, bet nepriklausantys nei *A*, nei *B*, yra sistemos būsenos klaidos, kurios kol kas nėra leidžiamos sistemai *P* su dabartiniais įvedimo duomenimis, bet gali būti priverstinai įvestos į *P,* pvz. klaidų įterpimo metodu. Atitinkamose situacijose, gali įvykti ir išorinė, ir vidinė klaida vienu metu.



2 pav. Klaidų klasės, kai aibės A ir B persidengia. A - vidinių klaidų aibė, B – išorinių klaidų aibė.

2 paveikslėlyje susikertančios aibės *A* ir *B* ($A \cap B$) vaizduoja klaidas, sukeltas netinkamos įeinančios informacijos ir vykdymo klaidų. Tai gali būti atvejis, kai blogi duomenys yra reikalingi išgauti tam tikru metodu vidinę klaidą. Ši pavaizduota situacija, kai $A \cap B$, reikalauja šiek tiek daugiau nei tradicinio testavimo. Norint simuliuoti *B* aibės klaidas įmanoma tik naudojantis klaidų įterpimo metodu. Tai reikalinga tam, kad būtų galima stebėti kokią įtaką išorinės klaidos (daviklių trikdžiai, sugadinta bylų informacija, žmogiškų klaidų faktorius) turės sistemai *P*. Norint simuliuoti $A \cap B$ aibės klaidas, reikia atlikti ir paprastą testavimą, ir naudoti klaidų įterpimo metodą.

Tarkime, kad aibę B galima padalinti į dvi dalis: b' ir b'' (2 paveikslėlis), kur b' yra klaidos, kurios tikrai įvyks sistemos P veikimo ateityje, o b'' – klaidos, kurios galėtų įvykti ateityje, bet iš tiesų neįvyks. Aibė b' yra baigtinis skaičius, o b'' greičiausiai yra begalinis daugumai P. b' ir b'' yra nežinomi prieš ir po klaidų įterpimo analizės vykdymo, todėl negalima žinoti, kuriai aibės B daliai šis narys priklauso. Reikia imituoti kiek įmanoma daugiau žinomų aibės B narių, nežinant kuriam poaibiui jie priklauso, ir tada atrinkti tiek kitų aibės Γ narių, kiek leidžia resursai.

2 paveikslėlyje pavaizduotas taškuotas apskritimas vaizduoja atsitiktiniu klaidų įterpimo metodu imituotas klaidas, pvz.: sugadintas vidines duomenų būsenas dalinai atsitiktiniu būdu. Dalinai, nes ta duomenų dalis, kuri yra sugadinta, yra jau sutaisyta. Pvz.: jei paskutinis vykdomas programinio kodo sakinys įtakojo kintamojo x reikšmę, tada sugadintume tik x duomenų būsenoje. Šis veiksmas imituoja x reikšmės skaičiavimų klaidą. Kintamojo x sugadinimo būdas yra ta vieta, kur įtraukiamas atsitiktinumo metodas. Jei paskutinis kodo sakinys įtakoja kontrolinį srautą, reiktų modifikuoti to sprendimo rezultatą, priverčiant rinktis alternatyvą (pasirinktinai ar atsitiktinai) kito sakinio vykdymo metu. Šis veiksmas imituoja sąlygos išraiškos klaidą. 2 paveikslėlyje pavaizduotas taškuotas apskritimas apima klaidas iš aibių A, b', b'' ir tų, kurios nepriklauso nei A, nei B. Būtų neteisinga, jei klaidų įterpimo metodas, naudojamas sistemos P, tiktų tik aibės b' narių imitavimui. Bet kokiu atveju, jei b' yra nežinomas, sėkmės kriterijaus praktiškai neįmanoma įgyvendinti, todėl reikia įterpti nepageidaujamas klaidas, naudojantis parametrų gadinimu, o ne pakeitimu. Kodas, kuris įvedamas vidiniams sakiniams modifikuoti, vadinamas perturbacijos, kitaip trikdymo, funkcija.

Kai kurie aibės B nariai gali būti žinomi, nes tam tikros klaidos vartotojams, išoriniams sistemos komponentams yra žinomos. Reikalavimai, kodas ir veikimo būdai yra ypatingai nestabilios būsenos. Kitaip nei kiti inžinerijos produktai, programinės įrangos sistemos yra nuolatinio kitimo būsenoje visą savo gyvavimo ciklą. Kai programinė įranga yra plėtojama, klaidų rizika auga ir darosi sunkiau nuspėti galimas jų atsiradimo vietas. Kiekvieną karta pakitus programinei įrangai reikėtų iš naujo atlikti klaidų įterpimo analizę ir pakartotinį testavimą.

Iki dabar teigėme, kad P yra konstanta. Jei P yra konstanta, aibė A liks fiksuota (tarkime, kad kodo ir duomenų įvesties reikalavimai nekinta). Klaidų rizikos valdymas turi būti itin lankstus susidūrus su P sistemos ir jos reikalavimų bei įvesties duomenų pokyčiais, tačiau svarbu žinoti, kaip aibė Γ gali pasikeisti, jei:

1. P ir jos reikalavimai nesikeičia, bet P duomenų įvesties reikalavimai keičiasi;

2. Kodo reikalavimai *P* sistemai keičiasi ir dėl to pati sistema *P* pakinta;
3. Kodo reikalavimai sistemai *P* nesikeičia, bet sistema *P* yra modifikuota bet kokiu atveju.

Pasikeitus kodui, Γ, *A*, *B*, *b'* ir *b''* aibės irgi pasikeis, o pokyčiai aibei *B* reikš, kad klaidų įterpimas turi būti atliktas iš naujo. Net jei sistema *P* yra pakeičiama tik tam, kad ištaisyti defektus, pasikeičia ir aibė *A*. Naujas kodas suteikia galimybę atsirasti naujiems defektams. Taigi, jei *P* pakinta, aibė *A* taipogi pasikeis. Naujiems defektams atrasti, į pagalbą pasitelkiamas regresinis testavimas, kurio metu pertestuojamas kodas su ankstesniais testavimo atvejais.

Tarkime, kad sistema veikia tam tikroje aplinkoje, vadinamoje operatyviniu profiliu (*O*). *O* aprašo dažnį, kuriuo tam tikos operatyvinės įvestys paduodamos į programinę įrangą. Dažnis, kuriuo aibės *A* nariai gali įtakoti *P* būsenas yra dalinai priklausomas nuo *O*. Kai klaidų įterpimas yra naudojamas blogiausio atvejo nuspėjimui, imituotos klaidos turi būti įterptos į sistemą *P*, kol *P* yra vykdoma priklausomai nuo *O*. Jei *O* yra realios veikimo aplinkybės, tai *O'* yra nerealios aplinkybės. Atliekant klaidų įterpimą, labai vertinga apgalvoti ir *O'*. Tai sumažintų galimų atvejų skaičių iš Γ, kai klaidos įvyktų, tik tada, kai kodas vykdomas su retais įvedimo duomenimis, retai pasitaikančias būdais.

Iš šio teorinio modelio matome, kad klaidų įterpimas yra procesas, kuris negali itin tiksliai pasakyti, kiek gera yra programinė įranga, tačiau jo pagalba galima nuspėti tam tikrus blogus sistemos veiksmus ateityje. Sistema bus laikoma labai gera tada, kai ji netoleruos dirbtinių klaidų, praktiškai nereaguos į šalutinius išorinius veiksmus ir veiks pagal reikalavimus.

2 SUDĖTINGŲSISTEMŲ MODELIAVIMAS IR FORMALIZAVIMAS

Sistema gali būti aprašyta kaip tam tikrų esybių rinkinys, kurios tarpusavyje sąveikauja. Sistemos sudėtingumas nusakomas nevienareikšmiškai, tačiau jį lemia pasirinktas detalumo lygis. Pagrindinės realaus laiko sistemų savybės:

1. Realaus laiko sistemos susideda iš didelio kiekio tarpusavyje sąveikaujančių esybių.
2. Sistemos elgesį lemia ne vienintelis valdiklis.
3. Žinant sistemos esybių elgseną, negalima daryti prielaidos, kad būtent taip elgis ir visa sistema.

Informatikoje formalizavimas apibrėžiamas kaip metodika, sukurta matematikos pagrindu, skirta programinės arba techninės įrangos sistemų specifikavimui, plėtojimui bei verifikavimui. Formalizmas yra reikšmingas didelėse integruotose sistemose, kuriose yra ypatingai svarbus saugumas ir patikimumas tam, kad būtų užtikrintas teisingas sistemos tobulinimo procesas bei išvengta klaidų atsiradimo. Formalizavimo metodai yra efektyviausi pačiose ankstyviausiose sistemos vystymo, reikalavimų ir specifikacijų dokumentų ruošimo stadijose, tačiau gali būti naudojami ir jau įgyvendintos sistemos formalizavimui.

Formalizavimo metodai gali būti naudojami tokiais būdais:

1. Formalus tobulinimas ir verifikavimas gali būti naudojamas kuriant sistemą šiek tiek formalesniais metodais. Pavyzdžiui, specifikacijose pateiktos ir įrodytos savybės vystymo metu yra perkeliamos į sistemos realizaciją. Toks naudojimo atvejis labiausiai tinka toms sistemoms, kuriose ypatingas dėmesys turi būti skiriamas saugumui arba patikimumui.
2. Sistema yra specifikuojama formaliai, tačiau tobulinama neformaliai tolesnėse stadijose. Toks panaudojimo atvejis reikalauja mažiausiai resursų ir duoda pakankamai gerus rezultatus.
3. Teoremų įrodymo priemonės gali būti panaudotos tam, kad būtų galima garantuoti visiškai formalias automatiškai patikrinamas tiesas. Tai vienas brangiausių panaudojimo atvejų, tačiau labiausiai atsiperkantis, ypatingai jeigu klaidų kainos sistemoje yra labai didelės (pvz. kritinių mikroprocesoriaus dalių projektavimas).

Formalizmo metodai yra skirstomi į grupes pagal savo semantines savybes ir pateikimo stilių:

- *Žymėjimo semantikos* grupei priklauso metodai, kuriais sistemos prasmė išreiškiama matematinės teorijos erdvėje. Šie metodai remiasi tuo, kad aprašinėjama sistema gali būti nesunkiai suprasta ir aprašyta matematinėje erdvėje, tačiau problema yra ta, kad ne kiekviena sistema intuityviai gali būti išreiškiama kaip funkcija.
- *Veikimo semantikos* grupei priklauso metodai, kuriais sistemos prasmė išreiškiama kaip modelio įvykių seka su paprastesniu skaičiavimu. Šios grupės metodų šalininkai pabrėžia, kad jų modeliai yra paprasti ir aiškūs, tačiau kritikai teigia, kad neišspręsta pačios semantikos problema. Modeliai sudaromi, tačiau pati semantika yra nepakankamai apibrėžta ir naudojant ją gali kilti problemų.
- *Aksiomatinei semantikai* priskiriami tokie metodai, kuriais sistemos prasmė išreiškiama išankstinėmis ir po to einančiomis sąlygomis, kurios yra atitinkamai teisingos prieš ir po tam tikros užduoties atlikimo.

2.1 Sudėtingų sistemų formalizavimo metodai

Kai kurie praktikai teigia, jog formalizavimo metodų visuma itin viršija pilnai sistemos specifikacijai ar modeliui aprašyti reikalingą formalizmą. Tvirtinama, kad aprašomosios kalbos išraiškingumas bei modeliuojamų sistemų sudėtingumas padaro visapusiško formalizavimo procesą labai sudėtingu ir reikalaujančiu didelių sąnaudų sprendimui uždaviniu. Buvo pasiūlyta alternatyva, kuriuose akcentuojamas dalinis formalizmas ir dėmesys nukreipiamas į jų pritaikomumą.

Pasaulyje dažniausiai naudojamos formalizavimo notacijos ir metodai:

- B-metodas
- Alloy
- Agentų modeliai
- Procesų algebra
- Luste
- Petri tinklai
- VDM
- Z
- Esterela

Alloy specifikacijos kalba paremta predikatų logika. Ji skirta nesudėtingos struktūros modeliams aprašyti. Alloy labiausiai pritaikyta programinės įrangos mikro modeliams aprašyti, tam kad būtų galima patikrinti jų teisingumą. Alloy specifikacijos kalba aprašytus modelius galima analizuoti ir patikrinti „Alloy Analyzer“ įrankiu. Šis įrankis generuoja modelio invariantinius atvejus, modeliuoja operacijų vykdymą ir gali patikrinti vartotojo nurodytas savybes. Galima teigti, kad Alloy yra Z notacijos poaibis. Kompoziciniu požiūriu Alloy mechanizmas turi Z notacijos lankstumą, tačiau remiasi šiek tiek kitokiomis idiomomis.

B-Metodas paremtas abstraktaus mechanizmo notacija, kuri dažnai naudojamas kuriant programinę įrangą. Metodo pradininkas – prancūzas Jean-Raymond Abrial. Kaip ir Alloy specifikacija, B yra susijęs su Z notacija ir gali būti naudojamas kuriant sistemas. Lyginant su Z – B-Metodas yra šiek tiek žemesnio lygmens ir daugiau dėmesio skiriama programinio kodo tikslumui, nei tiesiog formaliai specifikacijai. B-Metodas buvo naudojamas daugelyje kritinio saugumo reikalaujančiose sistemose (tokiose kaip Paryžiaus metro linijos) ir pramonėje susilaukė didelio dėmesio. Metodui taikyti sukurta galingų įrankių, padedančių specifikuoti, projektuoti, analizuoti ar generuoti programinį kodą.

Procesų algebra yra lygiagrečių sistemų formalaus modeliavimo metodų šeima. Nors egzistuoja didelė įvairovė procesų algebros atmainų, tačiau visos jos turi keletą bendrų požymių:

- vietoj visiems prieinamų kintamųjų modifikavimo, nepriklausomų procesų sąveikai pavaizduoti naudojami komunikacijų kanalai (žinutė – perdavimas);
- procesams ir sistemoms aprašyti naudojamos nedidelės primityvų kolekcijos bei operatoriai šiems primityvams sujungti;
- procesų operatoriams apibrėžiamos algebrinės tiesos, kurios leidžia manipuliuoti procesų išraiškomis naudojantis lygtimis.

Kanalų naudojimas komunikacijoms aprašyti yra viena iš savybių, išskiriančių procesų algebrą iš kitų lygiagretumą aprašančių modelių, tokių kaip Petri tinklai ar agentų modeliai.

Agentų modeliai yra matematinis lygiagrečių skaičiavimų modeliai sukurti 1973 m. Šiame modelyje agentu vadinamas lygiagrečių skaičiavimų primityvas: kaip atsaką į gautą žinutę, agentas gali atlikti vietinius sprendimus, sukurti daugiau agentų, išsiųsti daugiau žinučių ir nuspręsti, kaip atsakyti į kitą gautą žinutę. Agentų modelis gali būti naudojamas kaip karkasas lygiagretumo aiškinimui teoriniame lygmenyje bei kaip teorinis pagrindimas praktinių lygiagrečių sistemų realizacijoms.

Esterel yra vienalaikė programavimo kalba, skirta sudėtingoms sistemoms kurti. Esterel programavimo stilius leidžia nesunkiai išreikšti lygiagretumą, dėl ko ji labai tinka valdymo modeliams projektuoti. Kalba pradėta kurti 1980 m. Prancūzijoje. Šiandien Esterel kompiliatoriai gali transliuoti šia kalba parašytas programas ir generuoti C programinį kodą arba techninės įrangos aprašus (VHDL arba Verilog). Kalba vis dar tobulinama, nors jau sukurta keletas kompiliatorių (SCADE, Esterel Studio). Šios kalbos privalumai – tikslus programinis valdymas laike; patikimas lygiagretumo specifikavimas valdymo sistemoms; visiškai deterministiniai modeliai; baigtinių būsenų kalba (įmanoma nustatyti vykdymo laiką, gerokai paprastesnis formalus verifikavimas); gali būti naudojama tiek programinei, tiek techninei įrangai aprašyti. Esterel kalbos trūkumai – aprašinėjimas baigtinėmis būsenomis riboja lankstumą; sudėtingas duomenų apdorojimas; labiausiai tinkamas tik gana paprastiems sprendimus atliekantiems kontroleriams; sudėtingas kompiliavimo procesas.

Lustre – Esterel gimininga formalizavimo kalba, sukurta tų pačių kūrėjų, skirta sinchroniniais duomenų srautais valdomoms reaguojančioms sistemoms aprašyti. Pradėta kurti 1980 m., 1993 m. išsivystė į komercinį produktą, praktiškai naudojamą pramonėje. Dabar ji naudojama kritinėse valdymo sistemose – lėktuvuose, sraigtasparniuose ir atominėse elektrinėse.

Petri tinklai yra vienas iš keleto matematinių metodų diskrečiosioms paskirstytoms sistemoms aprašyti. Tai modeliavimo kalba, kuri orientuota grafo su paaiškinimais pavidalu, vaizduoja paskirstytos sistemos struktūrą. Petri tinklas turi viršūnes, perėjimo mazgus ir orientuotus lankus, jungiančius viršūnes su perėjimo mazgais. Petri tinklus 1962 m. išrado Carl Adam Petri. Bet kuriuo Petri tinklo vykdymo laiko momentu, bet kuri viršūnė gali turėti tam tikrą kiekį žymių. Priešingai nei tradicinėse duomenų apdorojimo sistemose, kurios gali apdoroti tik vieną srautą įėjimo žymių, Petri tinklų perėjimai gali apimti žymes iš keleto įėjimo vietų, juos atitinkamai apdoroti ir atiduoti žymes į skirtingas išėjimo viršūnes. Petri tinklai puikiai tinka lygiagrečiai paskirstytų sistemų modeliavimui.

VDM (Vienna Development Method) – sistemų kūrimo metodas, paremtas formalia specifikacija, parašyta VDM-SL specifikavimo kalba. Metodo naudojimui yra sukurti ir palaikomi įrankiai, taip pat sukurtas praplėtimas objektiškai orientuotam programavimui – VDM++. Metodas naudingas modeliais pagrįstoms sistemoms aprašinėti, tačiau netinkamas, jei sistema laikinė.

Z notacija - tai formali specifikavimo kalba, naudojama kompiuterinių sistemų aprašymui ir modeliavimui. Pagrindinis Z tikslas – išsami ir aiški kompiuterinės programos specifikacija ir numatytos programos veiksenos formalus įrodymas. Z išrasta 1970 m., Oksfordo universitete, ji remiasi aibių teorija ir predikatų logika. Z turi standartizuotą dažniausiai naudojamų matematinių funkcijų ir predikatų katalogą. Taip pat Z specifikacijose yra nemažai simbolių, nesančių ASCII kodų lentelėje, todėl specifikacijoje yra numatyti pasiūlymai, kaip koduoti Z notacijos simbolius ASCII kodų lentelės simboliais. Z notacija buvo praktiškai panaudota IBM CICS projekte

2.2 Sudėtingų sistemų imitacinis modeliavimas

Viena svarbiausių matematinio eksperimentavimo krypčių yra imitacinis modeliavimas. Apie imitacinį modeliavimą galima kalbėti tiek plačiąja, tiek ir siaurąja prasme.

Imitacinis modeliavimas siaurąja prasme – tai eksperimentavimas su jau paruoštais imitaciniais modeliais. Imitacinis modeliavimas plačiąja prasme – tai ne tik eksperimentai su imitaciniais modeliais, bet ir pačių imitacinių modelių sudarymas, prieš pradedant eksperimentus su jais, ir tų modelių tobulinimas eksperimentų metu. Imitacinis modeliavimas gali būti naudojamas ne tik kaip sistemų analizės, bet ir kaip jų veiklos optimizavimo priemonė, todėl galima kalbėti ne tik apie analitinius sisteminius modelius bei apie analitinius optimizacinius modelius, bet ir apie optimizavimo uždavinių sprendimą analitiniais metodais bei imitacinio modeliavimo priemonėmis.

Pagrindinis skirtumas tarp analitinių ir imitacinių sisteminių modelių yra tas, kad analitinių sisteminių modelių esmę sudaro matematinės formulės, išreiškiančios priklausomybę tarp objekto (sistemos) įėjimų, išėjimų ir būsenų vektorių koordinačių reikšmių – išėjimo ir perėjimo funkcijų matematinės išraiškos, o imitacinių sisteminių modelių esmę sudaro algoritmai, kuriais naudodamiesi galime imituoti objekto (sistemos) funkcionavimą, vidines ir išorines fazines, išėjimų bei įėjimų trajektorijas. Iš analitinių sisteminių modelių visuomet galima gauti imitacinius modelius, nes, turėdami perėjimo ir išėjimo funkcijas bei žinodami objekto (sistemos) pradinę būseną ir pradines įėjimo trajektorijų atkarpas, mes visuomet galime sudaryti fazinių ir išėjimo trajektorijų pradinių atkarpų apskaičiavimo algoritmus, o kartais – ne tik algoritmiškai, bet ir analitiškai išreikšti išėjimų bei fazinių trajektorijų priklausomybę nuo pradinės būsenos ir įėjimo trajektorijų. Iš imitacinių sisteminių modelių gauti analizinius modelius pavyksta gana retai, nes imitaciniai modeliai dažniausiai ir sudaromi kaip tik tais atvejais, kai mums nepavyksta nustatyti ir matematiškai išreikšti priklausomybių tarp įėjimų, išėjimų ir būsenų vektorių

koordinačių reikšmių. Ši aplinkybė sukelia ne tik imitacinių modelių trūkumus, lyginant su analitiniais, bet ir didelius jų pranašumus: imitacinius modelius galima kurti ir naudoti ten, kur nepavyksta sudaryti analitinių modelių. Mes, net nesugebėdami surasti ir matematiškai išreikšti priklausomybių tarp sudėtingos sistemos daugybės įėjimų, išėjimų ir būsenos vektoriaus koordinačių reikšmių, galime stebėti sistemos bei jos posistemių funkcionavimą ir, pasinaudodami pastebėtais to funkcionavimo dėsningumais, imituoti jį.

Ypač plačiai matematiniai imitaciniai modeliai ir imitacinio modeliavimo metodai taikomi nagrinėjant stochastines sistemas ir stochastinius procesus.

Sistemos, kurios kinta laike, tokios kaip transporto žiedinė sankryža kur automobiliai atvažiuoja ir išvažiuoja (kitaip vadinamos dinaminėmis sistemomis) yra susijusios su atsitiktiniais skaičiais. Niekas negali atspėti, kokiu tiksliu laiko momentu kitas automobilis įvažiuos į žiedinę sankryžą. Tokia sistema būtų puikus imitacinis modelis. Galima grafiškai apibrėžti tokios sistemos funkcionavimą, grafo viršūne žymėdami sankryžoje esančių automobilių skaičių (sistemos būsena). Atvykus automobiliui bei jam išvykstant atitinkamai kinta grafo struktūra. Toks grafas dar vadinamas galimų funkcionavimo trajektorijų grafu ir gali būti gaunamas tiek stebint realiai į žiedinę sankryžą įvažiuojančius automobilius, tiek dirbtinai modeliuojant sistemą. Tokį dirbtiną grafo konstravimą ir analizę apima imitacinis modeliavimas.

Galimi du modelių tipai: įvykiais valdomi ir tolydaus laiko modeliai. Įvykiais valdomuose modeliuose sistemos būsena kinta tik įvykus tam tikriems įvykiams ir tokių įvykių skaičius yra baigtinis. Tuo tarpu kituose modeliuose sistemos būsena gali kisti visą laiką, ne tik tam tikrais diskrečiais laiko momentais. Pavyzdžiui, vandens lygis rezervuare, į kurį gali būti pilamas, bei išleidžiamas vanduo. Tokiems atvejams modeliuoti kur kas tinkamesni tolydaus laiko modeliai.

Abiejų aukščiau minėtų modelių savybes apima agregatinis metodas.

2.2.1 Įvykiais valdomi modeliai

Įvykiais valdomais modeliais vadinami tokie modeliai, kuriuose sistemos būsena kinta tik diskrečiais (gali būti ir atsitiktiniais) laiko momentais. Sistemos būsena kinta tik tada, kai vyksta įvykiai, o laiko tarpai tarp įvykių sistemos būsenai įtakos neturi. Puikiu įvykiais valdomos sistemos pavyzdžiu gali būti detalių gamykla. Individualios esybės (detalės) yra renkamos pagal atitinkamus įvykius sistemoje (užsakymų numatymas ar priėmimas). Tarpai tarp įvykių šiuose modeliuose retai būna vienodi. Gana dažnai, tuo pačiu laiko momentu, reikia manipuliuoti su

dviem ar daugiau srauto elementais. Toks lygiagretus srauto valdymas galimas atliekant valdymą nuosekliai tuo pačiu laiko momentu. Dažniausiai tai sukelia logines problemas, kadangi iškyla elementų valdymo eiliškumo klausimas.

Dauguma kompiuterinių, loginių ir klaidų medžio modelių yra valdomi įvykiais. Čia nėra svarbus modeliavimas realiame laike. Kur kas svarbiau gauti modeliavimo rezultatus ir juos nagrinėjant aptikti logines projektavimo klaidas ar neteisingą įvykių nuoseklumą

2.2.2 Tolydaus laiko modeliai

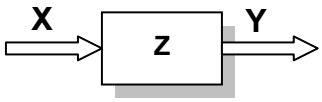
Tolydaus laiko modeliuose laiko eigoje sistemą apibūdinantys kintamieji gali kisti pastoviai, o ne tik tam tikrais diskrečiais laiko momentais. Šių kintamųjų reikšmės dažnai apibrėžiamos diferencialinėmis lygtimis. Diferencialinės lygtys šiuo atveju gali būti suprantamos kaip lygtys, reiškiančios ryšį tarp tolydžiojo kintamojo reikšmės ir jo kitimo koeficiento. Bendru atveju, tolydaus laiko modeliai susideda iš diferencialinių ir algebrinių lygčių rinkinio.

Bendru atveju agregatiniame metode tolydinės koordinatės gali būti aprašytos aukščiau minėtu būdu, o PLA atveju – tolydinė koordinatė $z_v(t)$ kinta pagal dėsnį

$$\frac{dz_v(t)}{dt} = -1.$$

3 AGREGATINIS FORMALIZAVIMO METODAS

Agregatinio sistemos specifikavimo požiūriu, sistema vaizduojama kaip tarpusavyje sąveikaujantys atkarpomis tiesiniai agregatai (*piece-linear agregate – PLA*) aibė. PLA suprantamas kaip objektas su apibrėžtų būsenų aibe Z, įėjimo signalų X ir išėjimo signalų Y aibėmis [7].



3 pav. Agregato struktūra

$X = \{x_1, x_2, \text{K}\};\ Y = \{y_1, y_2, \text{K}\};\ Z = \{z_1, z_2, \text{K}\}.$

Agregatinėje specifikacijoje visos tolydinės komponentės $w(e, t_m)$ yra keičiamos $w_e = 1$, ir jei operacija aktyvi, ir $w_e = 0$, jei operacija neaktyvi. Kadangi tolydinės komponentės nepriklauso nuo laiko, tai ir valdančios sekos tampa nereikalingos. Įvykdžius tokį pakeitimą sistemos būsena įgyja pavidalą:

$z = \{z_v; w_{e_1}, w_{e_2}, \text{K}, w_{e_n}\}$ ir yra prarandama laiko savybių analizės galimybė.

Agregato funkcionavimas yra analizuojamas laiko momentų aibėje $t \in T$. Būsena $z \in Z$, įėjimo signalas $x \in X$ ir išėjimo signalas $y \in Y$ laikomi laikinėmis funkcijomis. Be šių aibių taip pat turi būti apibrėžti perėjimo H ir išėjimo G operatoriai.

Atkarpomis tiesinio agregato būsena $z \in Z$ kiekvienu laiko momentu atitinka atkarpomis tiesinio Markovo proceso būseną:

$z(t) = (v(t), z_v(t))$, čia $v(t) = \{v_1(t), v_2(t), \text{K}, v_p(t)\}$ yra diskrečiosios būsenos komponentė, įgyjanti reikšmes iš baigtinių reikšmių aibės; ir $z_v(t)$ yra tolydžioji komponentė, susidedanti iš $w(e_1, t_m), w(e_2, t_m), \text{K}, w(e_k, t_m)$ koordinačių. Kiekviena iš šių komponenčių nusako, kada įvyks atitinkamas įvykis, galintis keisti agregato būseną.

Agregato būsena gali keistis tiktai dėl dviejų priežasčių: kai į agregatą ateina įėjimo signalas arba tolydinė komponentės reikšmė sutampa su einamu laiku. Įėjimo signalo priėmimo faktas vadinamas išoriniu įvykiu. Nesant įėjimo signalams, agregato būsena kinta pagal tokią

priklausomybę $v(t) = const, \frac{dz_v(t)}{dt} = -1$. Kiekvienam įvykiui $e_i'' \in E''$ apibrėžiama valdymo seka $\xi_0^{(i)}, \xi_1^{(i)}, \xi_2^{(i)}\text{K}$

Agregato funkcionavimas yra nagrinėjamas diskrečiais laiko momentais $T = \{t_1, t_2, \text{K}\ , t_m, \text{K}\}$, kuriais gali įvykti vienas ar keletas įvykių, iššaukiančių agregato būsenos pasikeitimą. Agregato būsenų aibė E susideda iš dviejų poaibių:

$E = E' \cup E''$. Į poaibį $E' = \{e_1', e_2', \text{K}\ , e_N'\}$ įeina įvykiai, sekantys iš įėjimo signalų aibės. E' įvykiai vadinami išoriniais įvykiais.

Poaibio $E'' = \{e_1'', e_2'', \text{K}\ , e_N''\}$ įvykiai vadinami vidiniais įvykiais. Perėjimo operatorius $H(e_i)$ nusako naują agregato būseną:

$$z(t_m) = H[z(t_m - 0), e_j],\ e_i \in E' \cup E''.$$

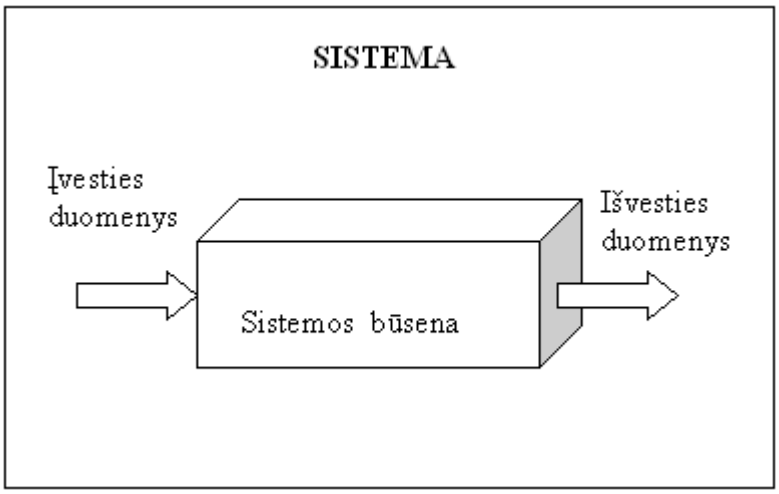
Išvedimo operatorius $G(e_i)$ apibrėžia išėjimo signalų turinį $y = G[z(t_m), e_i],\ e_i \in E' \cup E'',\ y \in Y$.

Paprastai PLA specifikacijos pateikiamos tokia forma:

1. Įėjimo aibė *X*.
2. Išėjimo aibė *Y*.
3. Išorinių įvykių aibė *E′*.
4. Vidinių įvykių aibė *E″*.
5. Valdančios sekos.
6. Diskreti būsenos dedamoji $v(t)$.
7. Tolydinė būsenos dedamoji $z_v(t)$.
8. Pradinė būsena $v(t_0)$, $z_v(t_0)$.
9. Perėjimo operatoriai *H, G*.

4 DEVS FORMALIZAVIMO METODAS

Imitacinis modeliavimas nėra vykdomas vien tik aprašant dinaminės sistemos struktūrą, tam taip pat yra reikalingas ir formalus sistemos aprašymas. Sistemos aprašymo formalizmas – tai sistemos specifikavimas, aprašant dinaminės sistemos elementų apribojimus. Toks aprašymas leidžia nustatyti lokalų dinaminės sistemos elgesį, t.y. kaip sistema elgsis bet kuriuo veikimo momentu [8].



4 pav. Sistemos koncepcinis modelis

4 paveikslėlyje pavaizduotas sistemos koncepcinis modelis. Sistema veikimo metu gali būti bet kurioje jai leidžiamoje būsenoje. Sistemai gavus tam tikrus įvesties duomenis, ji juos apdoroja, sistemos būsenos kinta, kol galiausiai išduodamas rezultatas – išvesties duomenys. Sistemos elgesį lemia įvesties ir išvesties duomenys, kuriuos sudaro duomenų įrašų poros, surinktos iš realių sistemų ar modelių, stebint ar atliekant eksperimentus.

Diskrečių įvykių sistemos specifikacija (*Discrete Event System Specification*) DEVS buvo sukurta Bernard Zeigler 1970m. Tai modulinis formalizmas, skirtas determinuotoms ir priežastinėms sistemoms, pritaikytas projektams, sukurtiems sudėtingų sistemų komponenčių pagrindu, aprašyti. DEVS formalizme svarbiausia yra specifikuoti sudėtinius modelius, iš kurių yra sudaryta sistema bei nustatyti ir aprašyti ryšius, kuriais sudėtiniai modeliai bendradarbiauja tarpusavyje. Dėl jiems būdingos diskretinės prigimties atominiai ir jungtiniai DEVS modeliai gali labai tiksliai ir efektyviai modeliuoti sistemas.

Egzistuoja labai daug klasikinio DEVS atmainų. Papildomi tam tikri elementai leidžia formalizuoti skirtingo tipo sistemas, stebint jų elgesį.

Pagrindinės DEVS atmainos:

- Dinaminės struktūros DEVS – sistemos imitavimo metu leidžia keisti modeliui savo struktūrą;
- Simbolinis DEVS – vaizduoja būsenų kitimą laike simbolinėje formoje, nustatant būsenų kitimo laike ryšius;
- Realaus laiko DEVS – suteikia galimybę DEVS modeliui būti kuriamam imitacinėje aplinkoje ir vykdomam labiau realiame laike nei imitaciniame;
- Miglotasis DEVS – sukurtos miglotos DEVS koncepcinės versijos, palengvinančios sistemos neapibrėžtumo aprašymą.

DEVS – tai pats bendriausias diskretinių sistemos įvykių modeliavimo formalizmas, leidžiantis aprašyti bet kurią sistemą, jei ji turi baigtinį skaičių būsenų pakitimų baigtiniuose laiko intervaluose. Tokiu atveju ne tik Petri tinklai, būsenų diagramos, įvykių grafai ir kitos diskretinės kalbos, bet ir visos diskretinės laiko sistemos gali būti kaip specifinis DEVS atvejis. Taip pat yra ir skaitmeniniai metodai QSS ir QSS2, vykdantys modeliavimą. Modeliavimo ir diskretinio specifinio įvykio sistemos modeliavimo formalizmo struktūra suteikia priemones matematinio modelio, atitinkančio sistemą, nustatymui. Sistema susideda iš laiko, įvesties duomenų, būsenų ir išvesties duomenų ir funkcijos tam, kad nustatyti būseną ir išėjimą, gautą keičiantis būsenai ir įėjimams. Diskretinio įvykio sistemos apibūdina kai kuriuos parametrus, kai veikia nepertraukiamos sistemos. Pavyzdžiui, diskretinių įvykio sistemų įėjimai įvyksta paskirstytu momentu, tuo pat laiku kaip ir nepertraukiamose sistemose yra nepertraukiamo laiko funkcijos. DEVS formalizmas apibūdinamas kaip diskretinė kalba įvykio modeliavimui, nustatanti diskretinius įvykių sistemos parametrus [5].

Norint formalizuoti sistemą DEVS metodu, reikalingos trys sudėtinės dalys:

- Reali sistema
- Modelis
- Imitatorius

Reali sistema yra pagrindinis duomenų šaltinis, veikianti pagal nustatytus reikalavimus. Modelis – visų sistemos instrukcijų bei duomenų, atitinkančių realią sistemą, rinkinys. Pagal juos yra sudaroma modelio elgsena. Imitatorius vykdo duotąsias modelio instrukcijas, skirtas elgsenos generavimui. Šie pagrindiniai objektai yra labai susiję tarpusavyje, nes jie apsprendžia, koks yra

modelio teisingumas. Modelis yra laikomas teisingu, jeigu duomenys, gauti iš modelio, atitinka duomenis, gautus iš realios sistemos eksperimento metu [8].

DEVS modelis turi standartinę struktūrą, kuri gali būti išplėsta tam tikrais parametrais ir naudojama skirtingose DEVS atmainose.

Klasikinės diskrečių įvykių sistemos specifikacijos (DEVS) struktūra:

$$M = [X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta]$$, kur:

X – įėjimo verčių sritis;

S – būsenų sritis;

Y – išėjimų verčių sritis;

$\delta_{\text{int}} : \text{S} \rightarrow \text{S}$ – vidinių perėjimų funkcija;

$\delta_{ext} : Q \times X \rightarrow S$ – išorinių perėjimų funkcija;

$Q = \{(s,e) | s \in S, 0 \le e \le ta(s)\}$ – pilna sistemos būsenų aibė;

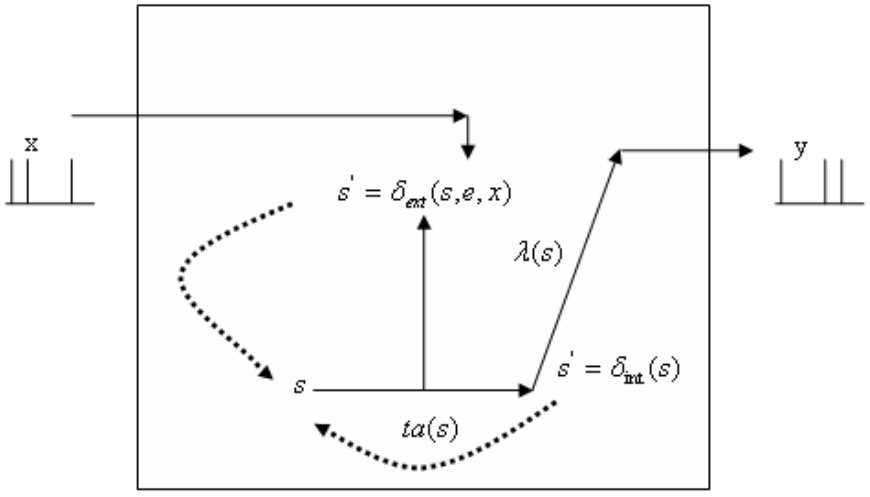
e – laikas, praėjęs nuo paskutinio perėjimo į kitą būseną;

$\lambda: S \rightarrow Y$ – išėjimo funkcija;

$ta: S \rightarrow \mathbf{R}_{+0,\infty}$ - laiko kitimo funkcija.

Šių elementų notacija yra pateikta 5 paveikslėlyje. Būtina paminėti, kad klasikinis DEVS modelis yra atominis DEVS, o atominiai DEVS modeliai gali būti sujungti į sudėtinį modelį.

Bet kuriuo laiku sistema yra tam tikroje būsenoje s. Jei sistemoje išorinis įvykis neįvyksta, sistema pasilieka būsenoje s nustatytą $ta(s)$ laiką ($0 \le ta(s) \le \infty$).



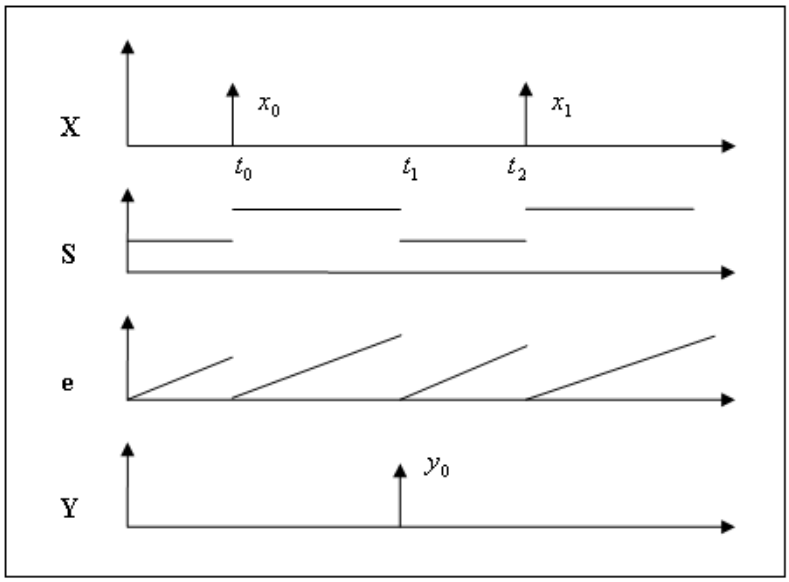
5 pav. DEVS būsenų kaita

Pirmuoju atveju sistemos buvimas būsenoje *s* yra toks trumpas, kad jokie išoriniai įvykiai negali jo įtakoti, todėl *s* yra trumpalaikė būsena.

Antruoju atveju sistema pasilieka būsenoje *s* visą likusį laiką. *s* šiuo atveju yra pasyvi būsena. Kai laukimo laikas baigiasi, t.y. kai baigtinis laikas *e=ta(s)*, sistemos išėjimo vertė *λ(s)*, o būsena pasikeičia į būseną $\delta_{int}(s)$. Reikėtų pažymėti, kad sistemos išvestis galima tik prieš vidinį būsenų perėjimą.

Jei išorinis įvykis $x \in X$ įvyksta prieš pasibaigiant nustatytam laikui, t.y. kai sistema yra būsenoje *(s,e)* su $e \le ta(s)$, sistema pereina į būseną $\delta_{ext}(s,e,x)$. Tokiu būdu vidinė perėjimo funkcija sukuria naują sistemos būseną, neįvykus jokiems įvykiams nuo paskutinio perėjimo. Išorinių perėjimų funkcija kuria naują sistemos būseną, kai įvyksta išorinis įvykis - ši būsena priklauso nuo įvesties duomenų, *x*, besikeičiančios būsenos, *s* ir to, kaip ilgai sistema buvo šioje būsenoje bei *e*, kai įvyko išorinis įvykis. Abiem atvejais sistema yra naujoje būsenoje *s'* su nauju nustatytu laukimo laiku *ta(s')* ir ta pati veiksmų seka kartojasi iš naujo.

Sistemos įvesties duomenimis yra laikomi tam tikri išoriniai įvykiai, tačiau pačiame modelyje jie yra neatsiejami ir nuo būsenų, laiko bei išvesties rezultatų. DEVS elgesys yra pavaizduotas 6 paveikslėlyje.



6 pav. DEVS trajektorijos

6 paveikslėlyje pavaizduotos DEVS trajektorijos, vaizduojančios, kaip tarpusavyje yra susiję išoriniai bei vidiniai įvykiai, sistemos buvimo tam tikroje būsenoje laikas bei išvesties duomenys. Išoriniai įvykiai įvyksta tam tikrais momentais, šiuo atveju *X* ašyje t_0 ir t_2 laiko

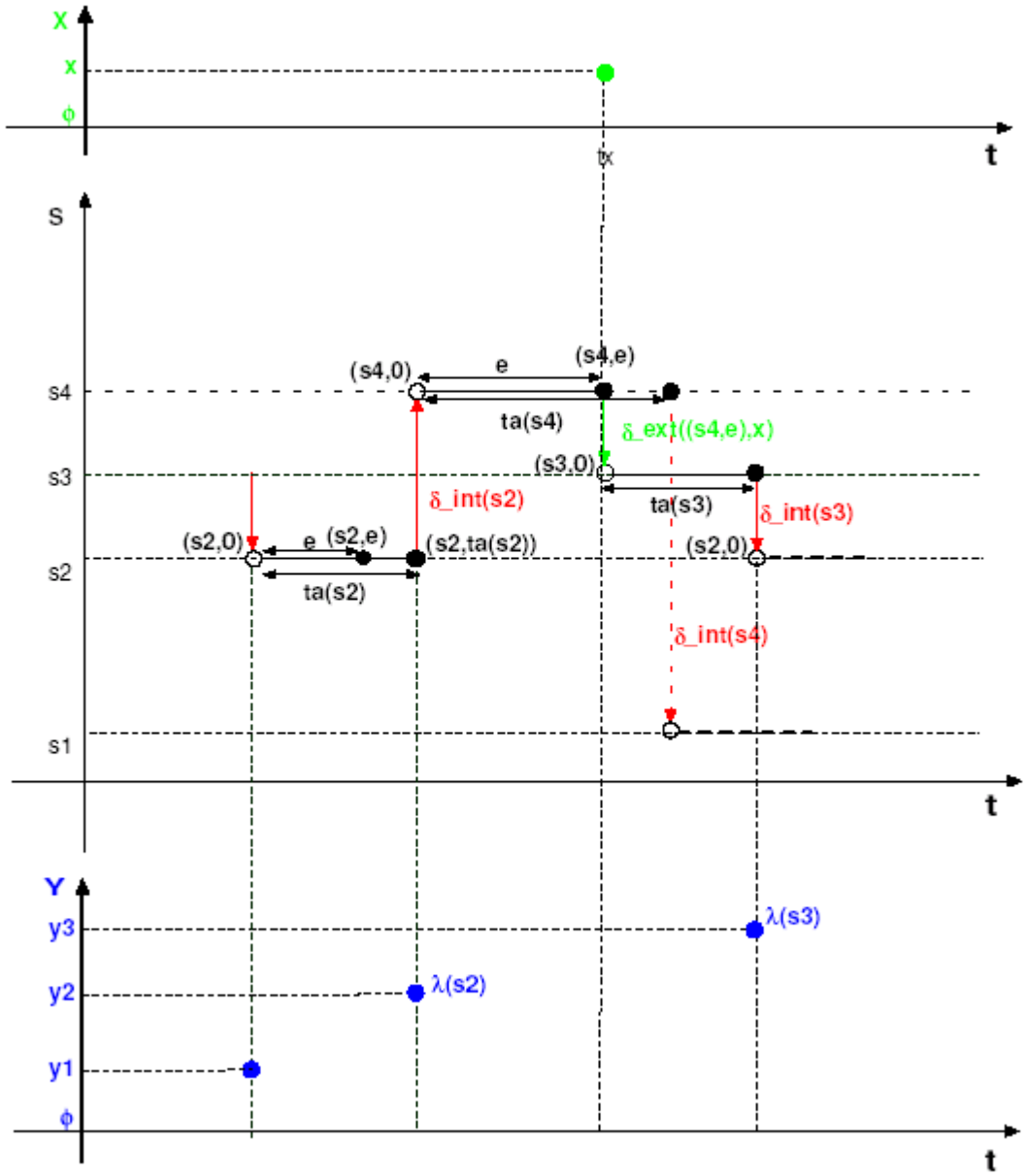
momentais, tačiau tarp šių įvykių gali įvykti ir vidiniai įvykiai (6 paveiksle jie pavaizduoti t_1 laiko momentu). Vidiniai įvykiai trajektorijų paveiksle atsispindi būsenų *S* ašyje. Praėjusio laiko *e* trajektorijoje parodoma, jog sistemai perėjus į kitą būseną, *e* laikmatis yra grąžinamas į nulinę reikšmę ir skaičiavimas prasideda iš naujo po kiekvieno įvykio. Žemiausioje *Y* ašyje vaizduojami išvesties duomenys, produkuoti išvesties funkcijos, prieš įvykstant vidiniam įvykiui [14].

DEVS modelyje laikas *t* yra tęstinis (=*R*). Būsenų aibė *S* yra leistina nuoseklių būsenų aibė: DEVS dinamika susideda iš tvarkingų S aibės būsenų. Dažniausiai - $S = \times_{i=1}^{n} S_i$. Tai formalizuoja keletą lygiagrečių sistemos dalių. Laikas, kai sistema pasilieka būsenoje prieš pereidama į sekančią būseną, yra modeliuojamas pagal išankstinio laiko (*time advance*) funkciją $ta: S \to \Re_{0,+\infty}^{+}$. Kadangi realiame pasaulyje laikas visada progresuoja, *ta* niekada negali būti neigiamas.

ta=0 leidžia parodyti momentinius perėjimus: nebūtina praeiti laiko tarpui, kad sistema galėtų pereiti į naują būseną. Jei sistema pasilieka galutinėje būsenoje s^* visą likusį laiką, tai modeliavimas vyksta pagal $ta(s^*) = +\infty$. Vidinė perėjimų funkcija $\delta_{\text{int}} : S \to S$ modeliuoja perėjimą iš vienos būsenos į kitą nuoseklią būseną. δ_{int} aprašo baigtinio būsenų automato elgseną, o *ta* papildo jį laiko eiga.

DEVS formalizuotoje sistemoje įmanoma stebėti sistemos išeigą. Išeigos aibė Y rodo leistinus išeigos rezultatus. Y yra struktūriškai apibrėžta aibė $Y = \times_{i=1}^{l} Y_i$. Ji formalizuoja išvesties prievadus.

DEVS formalizuotoje sistemoje nėra galimybės sugeneruoti išėjimą tiesiai iš išorinio įvykio. Išėjimas gali tik atsirasti prieš vidinį perėjimą. Išorinių Ryšys yra tarp išorinių perėjimų, vidinių perėjimų ir išėjimų pavaizduotas sekančiame lape esančiame 7 paveikslėlyje.



7 pav. Išorinių ir vidinių įvykių bei būsenų kaitos sąryšiai

7 paveikslėlyje matome, kad sistema atliko būsenos pakeitimą į būseną *s2*. Jei neįvyksta išorinis įvykis, sistema šioje būsenoje pasiliks nustatytą *ta(s2)* laiką. Per šį periodą praėjęs laikas *e* pasikeičia nuo *0* iki *ta(s2)*.

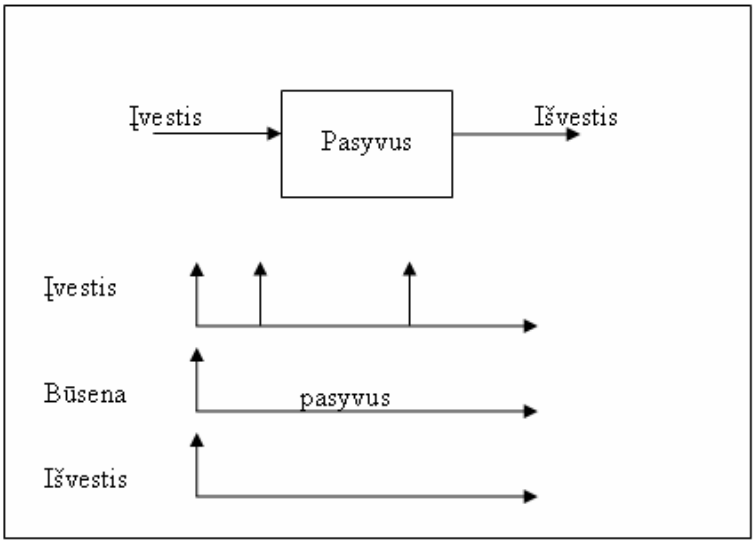
Kai laikas pasiekia *ta(s2)* vertę, pirmiausia išėjimo funkcija $\lambda(s2)$ sugeneruoja išėjimą – *y2,* ir sistema su vidinio perėjimo funkcija δ_{int} akimirksniu pereina į būseną s4 ($s4 = \delta_{\text{int}}(s2)$).

Automatinio režimo atveju, sistema būsenoje *s4* pasiliktų *ta(s4)* laiką ir jam pasibaigus pereitų į būseną $s1 = \delta_{\text{int}}(s4)$, prieš tai sugeneravus išėjimą. Tačiau paveikslėlyje prieš tai, kol *e* pasieka *ta(s4)* vertę, pasirodo išorinis įvykis *x*. Tuo metu sistema nebereaguoja į numatytą vidinį

perėjimą ir pereina į būseną $s3 = \delta_{\text{int}}((s4, e)x)$. Pasirodęs išorinis įvykis neturi įtakos išvesties duomenims. Ankstesnė būsena ir pasibaigęs laikas yra srityje (s,e), čia $s \in S$ ir $0 \le e \le ta(s)$.

Išvestis, sukurta perėjimo iš ankstesnės būsenos iki bet kurios kitos būsenos, gali būti išskaičiuotas su funkcija λ. Jeigu jo vertė yra Ø, reiškia, kad nėra nei vieno išėjimo.

Klasikiniu DEVS formalizuotos sistemos pavyzdys – sistema, sudaryta iš vieno komponento, kuris priima ir išduoda tam tikrus duomenis, tačiau nagrinėjamu atveju visada yra pasyvioje būsenoje ir nereaguoja į jokius įvykius.. Jos struktūra pavaizduota 8 paveiksle.



8 pav. Pasyvaus komponento struktūra

$DEVS = (X, Y, S, \delta_{ext}, \delta_{\text{int}}, \lambda, ta)$, kur:

$X = R$

$Y = R$

$S = \{"Pasyvus"\}$

$\delta_{ext} = ("Pasyvus", e, x) = "Pasyvus"$

$\delta_{\text{int}} = ("Pasyvus") = "Pasyvus"$

$\lambda("Pasyvus") = \varnothing$

$ta("Pasyvus") = \infty$

Šiame pavyzdyje X ir Y priklauso realių skaičių aibei. Sistemoje yra tik viena būsena „Pasyvus“. Šioje būsenoje sistema bus nustatytą *ta* laiką – t.y. nuolatos. Taip turėtų būti iki tol, kol sistemą pasieks įėjimo įvykis, tačiau formalizuotoje išraiškoje nurodoma, kad toks įvykis taip

pat bus ignoruojamas. Vidinės perėjimų ir išėjimų funkcijos formalizavimas iš esmės yra nereikalingas, nes jie niekada nebus vykdomi.

4.1 Atominis DEVS modelis

DEVS formalizavimo metodo esmė yra sistemos išskaidymas į smulkesnius sudėtinius modelius ir parodyti ryšius, kaip tie modeliai yra tarpusavyje susiję.

Smulkiausią DEVS komponentą, dar vadinamą atominiu modeliu, sudaro:

- Įvesties prievadų (*input port*) aibė, į kuriuos gaunami išoriniai įvykiai;
- Išvesties prievadų (output port) aibė,, iš kurių siunčiami išoriniai įvykiai;
- Sistemos kintamųjų ir parametrų aibė, dažniausiai pasitaiko šie du kintamieji – fazė ir sigma (nesant išoriniams įvykiams, sistema pasilieka esamoje fazėje nustatytą laika sigma);
- Vidinė perėjimo funkcija, kuri nurodo į kurią būseną sistema pereis po to, kai baigsis nustatytas laikas sigma;
- Išorinė perėjimo funkcija, nurodanti kaip sistema pakeis būseną, kai gaunami įvesties duomenys – pereis į naują fazę su duotąja sigma, taip pasiruošdama naujam vidiniam perėjimui;
- Išėjimo funkcija, generuojanti išorinius išvesties duomenis prieš įvykstant vidiniam perėjimui;
- Laiko progreso funkcija, kuri kontroliuoja vidinių perėjimų laiką.

4.2 Jungtinis DEVS modelis

Sujungti keli atominiai DEVS modeliai (komponentai) sudaro vieną jungtinį DEVS modelį. Jungtinio DEVS modelio semantika parodo, kaip bendradarbiauja keli į bendrą modelį sujungti atominiai komponentai. Jungtiniai modeliai gali būti patys dar kartą jungiami į aukštesnę hierarchiją ir sudaryti dar didesnį bei sudėtingesnį modelį.

Jungtinį DEVS modelį sudaro:

- Įvesties prievadų (*input port*) aibė, į kuriuos gaunami išoriniai įvykiai;
- Išvesties prievadų (output port) aibė, iš kurių siunčiami išoriniai įvykiai;
- Išorinių įvesties prievadų junginiai, skirti sujungti jungtinio modelio įvesties prievadą su vienu ar daugiau atominio modelio įvesties prievadų;

- Išorinių išvesties prievadų junginiai, sujungiantys jungtinio modelio įvesties prievadą su vienu ar daugiau atominio modelio įvesties prievadų – kai atominis modelis produkuoja išėjimo duomenis, šie duomenys gali būti perduoti jungtiniam modeliui ir taip tapti visos sistemos išvesties duomenimis;
- Intervalinės jungtys, kurios sujungia vieno atominio modelio įvesties prievadą su kito modelio išvesties prievadu – kai atominis modelis generuoja įvesties duomenis, jie gali būti siunčiami iš vieno atominio modelio į kitą, per vieno modelio įvesties tašką i kito modelio išvesties tašką.

Jungtinio DEVS modelio struktūra yra šiek tiek modifikuota nei atominio DEVS:

$D = [X, Y, N, M, I, Z, select]$, kur:

X - įvesties įvykių aibė;

Y - išvesties įvykių aibė;

N - komponentų vardų aibė, su apribojimu $D \notin N$;

$M = \{M_n \mid n \in N, M_n\}$ -modelis sudarytas iš vidinių smulkesnių modelių;

$I = \{I_n \mid n \in N, I_n \subseteq N \cup \{D\}\}$ - poveikių aibė kiekvienam n komponentui;

$Z = \{Z_{i,n} \mid \forall n \in N, i \in I_n Z_{i,n} : Y_i \to X_n \; arba \; Z_{D,n} : X \to X_n \; arba \; Z_{i,D} : Y_i \to Y\}$ - perėjimų funkcijos iš vieno komponento i į kitą komponentą n;

$select : 2^n \to N$ - pasirikimo funkcija.

4.3 FDEVS – DEVS atmaina su klaidų įterpimu

4 skyriaus pradžioje buvo minėta, kad DEVS formalizavimo metodas turi nemažai atmainų. Viena iš tokių atmainų yra FDEVS – DEVS formalizavimas su galimybe stebėti sistemos elgesį, įterpiant klaidas.Šį formalizmą sukūrė B. Zeigler, išplėsdamas standartines DEVS galimybes. Sistemą aprašant FDEVS metodu, galima valdyti tipines klaidų hipotezės klases – funkcines bei struktūrines. FDEVS formalizavimas apima ir atominį, ir jungtinį DEVS modeliavimo metodus kartu, todėl juo galima specifikuoti kompleksines sistemas. Klaidų imitavimas yra viena iš klaidoms tolerantiškų sistemos kūrimo dalių. Klaidos gali būti arba dirbtinai sukuriamos ir testuojamos, arba jau iš anksto numatomos specifikuojant kuriamos sistemos reikalavimus. Tai ypač svarbu kuriant kontrolines ar diagnostines sistemas [6].

Funkcinės klaidos modifikuoja DEVS modelio perėjimų funkcijas ir/arba išėjimo funkciją. Struktūrinės klaidos gali būti aprašomos jungtiniame DEVS modelyje. Darant atitinkamas prielaidas apie galimas ateityje sistemoje įvyksiančias klaidas, specifikuojamas klasikinis DEVS modelis.

Klasikinis DEVS modelis nuo FDEVS modelio skiriasi tuo, kad:

- Papildoma FDEVS yra įtraukiama ir klaidų aibė, kurią sudaro visos galimos klaidų hipotezės ir klaidų nebuvimo sąlyga. Ši klaidų aibė yra dalis perėjimų, išėjimų ir laiko funkcijų srities aprašo. Tai reiškia, kad šios funkcijos gali būti modifikuotos esant klaidai. Klaidų aibė taip pat susijusi su sistemos perėjimų funkcijomis- tai reiškia, kad modelis gali gyvuoti ir klaidos atveju;
- FDEVS turi modifikuotą funkciją – klaidų perėjimo, parodančią klaidos įterpimo efektą;
- DEVS neturi įvesties klaidų aibės, dėl kurių suveikia klaidų perėjimo funkcija.

Šis formalizavimo metodas bus panaudotas sekančiame darbo skyriuje, formalizuojant alternuojančio bito protokolą idealaus funkcionavimo atveju bei įvedus atitinkamus klaidų parametrus.

Tarkime, kad turime DEVS modelį:

$$M_n = \left\langle X_n, S_n, Y_n, \delta_{n-\text{int}}, \delta_{n-ext}, \lambda_n, ta_n \right\rangle$$

Norint įterpti klaidas, modelio aprašą reikia papildyti šiais parametrais:

$$M_f = \left\langle X, S, Y, \delta_{\text{int}}, \delta_{ext}, \delta_{fault}, \lambda, ta, X_f, F \right\rangle$$

$X = X_f \cup \tilde{X}$ yra įėjimo reikšmių aibė, kur $\tilde{X}$ yra naujų reikšmių aibė.

$S = S_n \cup \tilde{S}$ yra būsenų aibė, o $\tilde{S}$ yra naujų būsenų, į kurias modelis gali patekti esant klaidai, aibė.

$Y = Y_f \cup \tilde{Y}$ yra išėjimo reikšmių aibė, o $\tilde{Y}$ yra naujų išėjimo reikšmių, kurios gali būti produkuotos esant klaidai, aibė.

$F = F' \cup \{\phi\}$ yra klaidų aibė, kur F' yra visų galimu klaidų hipotezių aibė, o ϕ reiškia, jog klaidų visiškai nėra. Pažymėtina, kad vienas aibės F' elementas gali reikšti arba vieną, arba kelias klaidų hipotezes.

X_f yra įvykių aibė, kuri parodo klaidos atsiradimą arba išnykimą. Šie įvykiai yra valdomi modelio išorėje tam, kad būtų galima įterpti klaidas. Turint šį aprašymą, galima modeliuoti trumpalaikes arba nuolatines klaidas.

$\delta_{\text{int}} : S \times F \to S \times F$ yra vidinių perėjimų funkcija, turinti apribojimą $\delta_{\text{int}}(s, f) = (\delta_{n-\text{int}}(s), \phi)$.

$\delta_{ext} : S \times F \times R_0^+ \times X \to S \times F$ yra išorinių perėjimų funkcija, tenkinanti sąlygą $\delta_{ext}(s, \phi, e, x) = (\delta_{n-ext}(s), \phi)$ jei $x \in X_n$.

$\delta_{fault} : S \times F \times R_0^+ \times X_f \to S \times F$ yra klaidos perėjimo funkcija.

$\lambda : S \times F \to Y$ yra išėjimo funkcija, verifikuojanti apribojimą $\lambda(s, \phi) = \lambda_n(s)$.

$ta : S \times F \to R_0^+$ yra gyvavimo ciklo funkcija, turinti apribojimą $ta(s, \phi) = ta_n(s)$.

FDEVS vykdymo semantika yra tokia pati, kaip ir DEVS. Skirtumas yra tas, kad bet kuriuo metu pasirodo klaida, vyksta perėjimas, kuris suskaičiuojamas pagal perėjimo funkciją. Esant išoriniams įvykiams, klaida neprodukuos jokio išėjimo.

Pavyzdys: tarkime, kad procesorius gali atlikti kelis darbus: $\{job_1, job_2, ..., job_n\}$. Nesant klaidai, kiekvienas darbas yra atliekamas per laiką $t_p(job_i)$. Kai procesorius baigia darbą, gaunamas išėjimas $y(job_i)$. Kol procesorius vykdo darbą, bet kuris kitas atėjęs darbas yra ignoruojamas. Nustačius šias sąlygas, sudaromas DEVS modelis [6]:

$M_n = \langle X_n, S_n, Y_n, \delta_{n-\text{int}}, \delta_{n-ext}, \lambda_n, ta_n \rangle$

$X_n = \{job_1, job_2, ..., job_n\}$ t.y. kad procesorius gali atlikti n darbų.

$S_n = \{job_1, job_2, ..., job_n\} \cup \{\phi\} \times R_0^+$

$Y_n = \{y(job_1), y(job_2), ..., y(job_n)\}$

$\delta_{n-\text{int}}(job, \sigma) = (\phi, \infty)$ tai reiškia, kad joks darbas nėra aptarnaujamas

$$\delta_{n-ext}(job, \sigma, e, x) = \begin{cases} (x, t_p(x)) & if \ job = \phi \\ (job, \sigma - e) & otherwise \end{cases}$$

$\lambda_n(job, \sigma) = y(job)$

$\lambda_n(job, \sigma) = \sigma$

Aprašius tradicinį modelį, iškeliamos dvi skirtingos hipotezės:

1. Jei procesorius aptarnauja paraišką (*job*), atvykus naujai paraiškai, antroji paraiška nebebus ignoruojama. Pirmosios paraiškos aptarnavimas bus nutrauktas, o pradėta aptarnauti antroji paraiška.
2. Aptarnaujant vieną paraišką ir atėjus antrajai, aptarnaujamos bus abi paraiškos, todėl procesorius dirbs tik puse savo pajėgumo.

Abi klaidos gali atsirasti tuo pačiu metu, tai vadinama daugybine klaidų hipoteze. Pirmoji klaida bus įterpiama pavadinimu *BFon*., o antroji – *SFon*. Atitinkamai *BFoff* ir *SFoff* parametrai bus naudojami klaidoms pašalinti iš sistemos.

Pagal atitinkamus pokyčius, modelio struktūra taps tokia:

$$M_n = \left\langle X, S, Y, \delta_{\text{int}}, \delta_{ext}, \delta_{fault}, \lambda, ta, X_f, F \right\rangle$$

$X = X_n$ - įvesties duomenys;

$S = S_n$ - būsenų aibė;

$Y = Y_n$ - išvesties duomenys;

$X_f = \{BFon, SFon, BFoff, SFoff\}$ - galimos klaidų būsenos;

$F = \{BF, SF, BFSF, \phi\}$- klaidų reikšmės (pirmo tipo, antro tipo, abiejų tipų, klaidų nėra);

$\delta_{\text{int}}\left((job, \sigma) f\right) = \left((\phi, \infty), f\right)$ - jokia paraiška nėra aptarnaujama;

$$\delta_{ext}(job, \sigma, e, x, f) = \begin{cases} (\delta_{n-ext}(job, \sigma, e, x), f) & if \quad f = \phi \\ ((job, \sigma - e), f) & if \quad f = SF \wedge job \neq \phi \\ ((x, t_1), f) & otherwise \end{cases}$$

$$where \begin{cases} tp(x) if\ f = BF \\ 2tp(x) otherwise \end{cases}$$

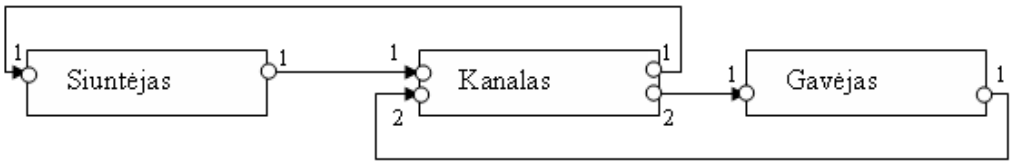
$$\delta_{fault}(job, \sigma, e, f, x_f) = \begin{cases} ((job, \sigma - e), BF) if\ f = \phi \wedge x_f = BFon \\ ((job, \sigma - e), BFSF) if\ f = SF \wedge x_f = BFon \\ ((job, \sigma - e), \phi) if\ f = BF \wedge x_f = BFoff \\ ((job, \sigma - e), SF) if\ f = BFSF \wedge x_f = BFoff \\ ((job, 2(\sigma - e)), SF) if\ f = \phi \wedge x_f = SFon \\ ((job, 2(\sigma - e)), BFSF) if\ f = BF \wedge x_f = SFon \\ \left(\left(job, \frac{(\sigma - e)}{2}\right), \phi\right) if\ f = SF \wedge x_f = SFoff \\ \left(\left(job, \frac{(\sigma - e)}{2}\right), BF\right) if\ f = BFSF \wedge x_f = SFoff \\ ((job, \sigma - e), f) otherwise \end{cases}$$

$$\lambda_n(job, \sigma, f) = \lambda_n(job, \sigma)$$

$$ta(job, \sigma, f) = ta_n(job, \sigma)$$

5 ALTERNUOJANČIO BITO PROTOKOLO FORMALIZAVIMAS

Alternuojančio bito protokolas (*alternating bit protocol*) ABP – tai kompiuterinis protokolas, skirtas užtikrinti patikimą duomenų perdavimą tinkle. Paketai perduodami pusiau dupleksinį kanalą. Protokolo schema pavaizduota 9 paveiksle [3]:



9 pav. Alternuojančio bito protokolo schema

Siuntėjas siunčia paketą ir laukia patvirtinimo, kad paketą gavėjas priėmė. Tam, kad atskirti du iš eilės einančius paketus, siuntėjas prideda papildomą bitą ant kiekvieno paketo (vadinamąjį alternuojantį bitą, nes siuntėjas naudoja arba 0, arba 1).

Šiame darbe alternuojančio bito protokolas yra nagrinėjamas dviem aspektais:

1. Protokolo funkcionavimas idealiomis sąlygomis, darant prielaidą kad bet kuriuo atveju paketas bus sėkmingai perduodamas gavėjui.
2. Protokolo funkcionavimas esant tikimybei, kad įvyks klaida, sutrikdanti teisingą alternuojančio bito protokolo veikimą.

5.1 ABP agregatinė specifikacija idealiu funkcionavimo atveju

5 skyriaus pradžioje buvo minėta, kad alternuojančio bito protokolas pirmiausiai aprašomas idealiu atveju, laikant kad bet kuris siuntėjo išsiųstas paketas bus teisingas ir iš karto priimtas gavėjo. Protokolo funkcionavimas idealiu atveju gali būti tik teorinis modelis, nes neegzistuoja tokia reali sistema, kurioje nebūtų sistemos klaidos tikimybės. Tokiu atveju, neįmanoma garantuoti, kad protokolas visada veiks teisingai.

Tam, kad aprašyti alternuojančio bito protokolą agregatiniu metodu, būtina jį išskaidyti į agregatus – Siuntėją, Gavėją ir Kanalą. Šie agregatai yra formalizuojami pagal 3 šio darbo skyriuje aprašytas agregatinio metodo formalizavimo taisykles.

5.1.1 Agregato Siuntėjas specifikacija

1. **Įėjimo signalų aibė:**

X={x_1}, x_1= B,

B – alternuojančio bito reikšmė patvirtinime, B $\in$ {0,1}

2. Išėjimų signalų aibė:

Y={y_1}, y_1=B,

B – alternuojančio bito reikšmė perduodame pakete, B $\in$ {0,1}

3. Išorinių įvykių aibė:

E‘={e'_{11}},

e'_{11} - priimtas signalas x_1

4. Vidinių ivykių aibė:

E‘‘= {e''_{11}},

e''_{11} - paketas suformuotas siųsti

5. Valdymo sekos e''_{11} α {η_{1j}}$_{j=1}^{\infty}$,

η_{1j} - j-ojo paketo formavimo trukmė

6. Diskrečioji būsenos dedamoji:

(t_m)={PSK(t_m), $Bit_1(t_m)$},

PSK(t_m) – priimtų patvirtinimų skaičius

$Bit_1(t_m)$ - alternuojančio bito reikšmė paskutiniame išsiųstame pakete

7. Tolydžioji būsenos dedamoji z_v (t_m)={$w(e''_{11}, t_m)$),

$w(e''_{11}, t_m)$ **-** eilinio paketo formavimo pabaigos momentas

8. Parametrai

$\varnothing$ - idealaus protokolo funkcionavimo atveju papildomi parametrai neįvedami;

9. Pradinė būsena

z(t_0)={0,1, η_{11}, ∞}

10. Perėjimo ir išėjimo operatoriai:

H(e'_{11}): /Priimtas signalas x1/

if ($B = Bit_1(t_m)$ then /jeigu alternuojančio bito reikšmė patvirtinime sutampa su alternuojančio bito reikšme paskutiniame išsiųstame pakete

$Bit_1(t_{m+1}) = \overline{Bit_1(t_m)}$ /invertuojama bito reikšmė/

PSK(t_{m+1})=PSK(t_{m+1})+1 /priimtų patvirtinimų skaičius padidinimas vienetu/

$w(e''_{11}, t_{m+1}) = t_m + \eta_{1j}$

else

$w(e''_{11}, t_{m+1}) = t_m + \eta_{1j-1}$

$H(e''_{11})$: /Paketas suformuotas siųsti/

$w(e''_{11}, t_{m+1}) = \infty$

$G(e''_{11})$: $y_1 = (Bit1(t_m))$

5.1.2 Agregato Kanalas specifikacija

1. **Įėjimo signalai:**

x_1=(B), x_2 = (B),

B – alternuojančio bito reikšmė perduodamame pakete ar patvirtinime: signalas x_1 perduodamas iš siuntėjo, o signalas x_2 iš gavėjo.

2. **Išėjimo signalai:**

y_1=(B), y_2 =(B),

B – alternuojančio bito reikšmė perduodamame pakete ar patvirtinime

3. **Išorinių įvykių aibė**

E‘=$\{e'_{21}, e'_{22}\}$,

e'_{21} - signalas x_1 atėjo iš Siuntėjo;

e'_{22} - signalas x_2 atėjo iš Gavėjo

4. **Vidinių įvykių aibė:**

E‘‘=$\{e''_{21}, e''_{22}\}$,

e''_{21} - patvirtinimo perdavimas baigėsi

e''_{22} - paketo perdavimas kanalu baigėsi

5. **Valdymo sekos:**

e''_{21} α $\{\xi_{1j}\}_{j=1}^{\infty}$, e''_{22} α $\{\xi_{2j}\}_{j=1}^{\infty}$

ξ_{1j} - j-tojo paketo patvirtinimo perdavimo trukmė

ξ_{2j} - j-tojo paketo perdavimo trukmė

6. **Agregato būsena:**

v(t_m)= { $Bit2(t_m)$ }, $z_v(t_m) = \{w(e''_{21}, t_m), w(e''_{22}, t_m)\}$

Bit2(t_m) – alternuojančio bito reikšmė perduodame pakete/patvirtinime

$w(e''_{21}, t_m)$ - patvirtinimo perdavimo kanalu pabaigos momentas

$w(e''_{22}, t_m)$ - paketo perdavimo kanalu pabaigos momentas

7. Parametrai:

$\varnothing$ - idealaus protokolo funkcionavimo atveju papildomi parametrai neįvedami;

8. Perėjimo ir išėjimo operatoriai:

H(e'_{21}): / Signalas x_1 atėjo iš Siuntėjo/

if ($w(e''_{21}, t_m) = \infty$) $\wedge$ ($w(e''_{22}, t_m) = \infty$) then

Bit2 (t_{m+1})=B

fi

H(e'_{22}): / Signalas x_1 atėjo iš Gavėjo /

if ($w(e''_{21}, t_m) = \infty$) $\wedge$ ($w(e''_{22}, t_m) = \infty$) then

Bit2 (t_{m+1})=B

fi

H(e'_{21}): / Baigėsi patvirtinimo perdavimas /

$w(e''_{21}, t_{m+1}) = \infty$

G(e''_{21}): y_1=(Bit2(t_{m+1}))

H(e''_{22}): / Baigėsi paketo perdavimas kanalu/

$w(e''_{22}, t_{m+1}) = \infty$

G(e''_{21}): y_2 =(Bit2(t_{m+1}))

5.1.3 Agregato Gavėjas specifikacija

1. Įėjimo signalai:

x_1=(B)

B – alternuojančio bito reikšmė priimame pakete

2. Išėjimo signalai:

y_1=(B)

B – alternuojančio bito reikšmė perduodamame patvirtinime

3. **Išorinių įvykių aibė**:

$E^{'} = \{e_{31}^{'}\}$

$e_{31}^{'}$ - signalo x_1 atėjimas

4. Vidinių įvykių aibė:

$E^{''} = \{e_{31}^{''}\}$

$e_{31}^{''}$ - patvirtinimo formavimo pabaiga

5. Valdymo sekos:

$e_{31}^{''}$ α $\{\xi_{3j}\}_{j=1}^{\infty}$

ξ_{3j} -j-ojo patvirtinimo formavimo trukmė

6. Agregato būsena

v(t_m)= { $KSK(t_m), Bit3(t_m)$ }, $z_v(t_m) = \{w(e_{31}^{''}, t_m)\}$

$Bit3(t_m)$ - alternuojančio bito reikšmė paskutiniame suformuotame patvirtinime

$KSK(t_m)$ - priimtų paketų skaičius

$w(e_{31}^{''}, t_m)$ - laiko momentas, kai baigsis patvirtinimo formavimas

7. Parametrai:

$\varnothing$ - idealaus protokolo funkcionavimo atveju papildomi parametrai neįvedami;

8. Pradinė būsena:

$z(t_0) = \{0, \infty, \infty\}$

9. Perėjimo ir išėjimo operatoriai:

H($e_{31}^{'}$): /Signalas x_1 atėjo /

if $w(e_{31}^{''}, t_m) = \infty$ then

KSK(t_{m+1})=KSK (t_m)+1 /priimtų paketų skaičius padidinamas vienetu/

Bit3 (t_{m+1})=B

$w(e_{31}^{''}, t_{m+1}) = t_m + \eta_{3j}$

else

$w(e_{31}^{''}, t_{m+1}) = t_m + \eta_{3,j-1}$

fi

H(e''_{31}): / Patvirtinimo formavimo pabaiga/

$w(e''_{31}, t_{m+1}) = \infty$

G(e''_{31}): y_1=(Bit3(t_{m+1}))

5.2 Agregatinė tolerantiško klaidoms ABP specifikacija

Šio darbo 5.1 dalyje buvo specifikuotas alternuojančio bito protokolo(ABP) idealiu funkcionavimo atveju teorinis modelis. Tačiau norint, kad šis protokolas iš tiesų teisingai veiktų realybėje, reikia įvertinti ir galimas klaidas, kurios sutrikdytų šio protokolo darbą ir tų klaidų buvimo atveju jis nesugebėtų teisingai funkcionuoti [3].

Dažniausiai pasitaikančios klaidos yra perduodamo paketo iškraipymai bei jų pametimai. Specifikuojant šį protokolą neidealiu atveju, šios klaidos bus įvertinamos.

5.2.1 Agregato Siuntėjas specifikacija

1. **Įėjimo signalų aibė:**

X={x_1}, x_1= B,

B – alternuojančio bito reikšmė patvirtinime, B $\in$ {0,1}

2. **Išėjimų signalų aibė:**

Y={y_1}, y_1=B,

B – alternuojančio bito reikšmė perduodame pakete, B $\in$ {0,1}

3. **Išorinių įvykių aibė:**

E‘={e'_{11}},

e'_{11} - priimtas signalas x_1

4. **Vidinių įvykių aibė:**

E‘‘={e''_{11}, e''_{12}},

e''_{11} - paketas suformuotas siųsti

e''_{12} - baigėsi laikmačiui nustatytas laikas (timeout)

5. **Valdymo sekos** e''_{11} α $\{\eta_{1j}\}_{j=1}^{\infty}$, e''_{12} α $\{\tau_{1j}\}_{j=1}^{\infty}$

η_{1j} - j-ojo paketo formavimo trukmė

τ_{1j} = const – laikmačiui nustatytas laikas

6. Diskrečioji būsenos dedamoji:

v(t_m)={PSK(t_m), $Bit_1(t_m)$ }

PSK(t_m) – priimtų patvirtinimų skaičius

$Bit_1(t_m)$ - alternuojančio bito reikšmė paskutiniame išsiųstame pakete

7. Tolydžioji būsenos dedamoji $z_v(t_m)$={$w(e''_{11}, t_m)$, $w(e''_{12}, t_m)$}

$w(e''_{11}, t_m)$ - eilinio paketo formavimo pabaigos momentas

$w(e''_{12}, t_m)$ - laikmačiui nustatytas pabaigos momentas

8. Parametrai

P_{11} - patvirtinimo klaidingo perdavimo tikimybe

RND(1) – atsitiktinė reikšmė, tolygiai pasiskirsčiusi intervale [0,1]

9. Pradinė būsena

z(t_0)={0,1, η_{11}, ∞ }

10. Perėjimo ir išėjimo operatoriai:

H(e'_{11}): /Priimtas signalas x1/

if (B= $Bit_1(t_m)$) $\wedge$ (P_{11} < RND(1)) then

$Bit_1(t_{m+1}) = \overline{Bit_1(t_m)}$

PSK(t_{m+1})=PSK(t_{m+1})+1

$w(e''_{11}, t_{m+1}) = t_m + \eta_{1j}$

else

$w(e''_{11}, t_{m+1}) = t_m + \eta_{1j-1}$; /Paketo formavimas baigtas/

$w(e''_{12}, t_{m+1}) = \infty$;

H(e''_{11}): /Paketas suformuotas siųsti/

$w(e''_{11}, t_{m+1}) = \infty$

$w(e''_{12}, t_{m+1}) = t_m + \tau_i$

G(e''_{11}):

$y_1 = (Bit1(t_m))$; /Išduodama alternuojančio bito reikšmė/

$H(e_{12}^{"})$: /Baigėsi laikmačiui nustatytas laikas/

$w(e_{12}^{"}, t_{m+1}) = t_m + \eta_{1j-1}$

$w(e_{12}^{"}, t_{m+1}) = \infty$

5.2.2 Agregato Kanalas specifikacija

1. **Įėjimo signalai:**

x_1=(B), x_2 = (B),\

B – alternuojančio bito reikšmė perduodamame pakete ar patvirtinime: signalas x_1 perduodamas iš siuntėjo, o signalas x_2 iš gavėjo.

2. **Išėjimo signalai:**

y_1=(B), y_2 =(B),

B – alternuojančio bito reikšmė perduodamame pakete ar patvirtinime

3. **Išorinių įvykių aibė**

E‘=$\{e_{21}^{'}, e_{22}^{'}\}$,

$e_{21}^{'}$ - signalas x_1 atėjo iš Siuntėjo;

$e_{22}^{'}$ - signalas x_2 atėjo iš Gavėjo

4. **Vidinių įvykių aibė:**

E‘‘=$\{e_{21}^{"}, e_{22}^{"}\}$,

$e_{21}^{"}$ - patvirtinimo perdavimas baigėsi $e_{22}^{"}$ - paketo perdavimas kanalu baigėsi

5. **Valdymo sekos:**

$e_{21}^{"}$ α $\{\xi_{1j}\}_{j=1}^{\infty}$, $e_{22}^{"}$ α $\{\xi_{2j}\}_{j=1}^{\infty}$

ξ_{1j} - j-tojo paketo patvirtinimo perdavimo trukmė

ξ_{2j} - j-tojo paketo perdavimo trukmė

6. **Agregato būsena:**

v(t_m)= $\{Bit2(t_m)\}$, $z_v(t_m) = \{w(e_{21}^{"}, t_m), w(e_{22}^{"}, t_m)\}$

Bit2(t_m) – alternuojančio bito reikšmė perduodame pakete/patvirtinime

$w(e_{21}^{"}, t_m)$ - patvirtinimo perdavimo kanalu pabaigos momentas

$w(e_{22}^{"}, t_m)$ - paketo perdavimo kanalu pabaigos momentas

7. **Parametrai:**

P_{21} - paketo praradimo tikimybė kanale;

P_{22} - patvirtinimo praradimo tikimybė kanale;

RND(2) – atsitiktinis dydis, tolygiai pasiskirstęs intervale [0,1]

8. Perėjimo ir išėjimo operatoriai:

H($e_{21}^{'}$): / Signalas x_1 atėjo iš Siuntėjo/

if ($w(e_{21}^{''}, t_m) = \infty$) $\wedge$ ($w(e_{22}^{''}, t_m) = \infty$) then

Bit2 (t_{m+1})=B

if P_{21} <RND(2) then $w(e_{22}^{''}, t_{m+1}) = \infty$

else $w(e_{22}^{''}, t_{m+1}) = t_m + \xi_{2j}$

fi

fi

H($e_{22}^{'}$): / Signalas x_1 atėjo iš Gavėjo /

if ($w(e_{21}^{''}, t_m) = \infty$) $\wedge$ ($w(e_{22}^{''}, t_m) = \infty$) then

Bit2 (t_{m+1})=B

if P_{21} <RND(2) then $w(e_{21}^{''}, t_{m+1}) = \infty$

else $w(e_{21}^{''}, t_{m+1}) = t_m + \xi_{1j}$

fi

fi

H($e_{21}^{'}$): / Baigėsi patvirtinimo perdavimas /

$w(e_{21}^{''}, t_{m+1}) = \infty$

G($e_{21}^{''}$): y_1=(Bit2(t_{m+1}))

H($e_{22}^{''}$): / Baigėsi paketo perdavimas kanalu/

$w(e_{22}^{''}, t_{m+1}) = \infty$

G($e_{21}^{''}$): y_2 =(Bit2(t_{m+1}))

5.2.3 Agregato Gavėjas specifikacija

1. Įėjimo signalai:

x_1=(B)

B – alternuojančio bito reikšmė priimame pakete

2. Išėjimo signalai:

y_1=(B)

B – alternuojančio bito reikšmė perduodamame patvirtinime

3. Išorinių įvykių aibė:

$E' = \{e'_{31}\}$

e'_{31} - signalo x_1 atėjimas

4. Vidinių įvykių aibė:

$E'' = \{e''_{31}\}$

e''_{31} - patvirtinimo formavimo pabaiga

5. Valdymo sekos:

e''_{31} α $\{\xi_{3j}\}_{j=1}^{\infty}$

ξ_{3j} -j-ojo patvirtinimo formavimo trukmė

6. Agregato būsena

v(t_m)= $\{ KSK(t_m), Bit3(t_m) \}$, $z_v(t_m) = \{w(e''_{31}, t_m)\}$

$Bit3(t_m)$ - alternuojančio bito reikšmė paskutiniame suformuotame patvirtinime

$KSK(t_m)$ - priimtu paketų skaičius

$w(e''_{31}, t_m)$ - laiko momentas, kai baigsis patvirtinimo formavimas

7. Parametrai:

P_{31} - paketo klaidingo perdavimo tikimybė

RND(1) – atsitiktinis dydis, tolygiai pasiskirstęs intervale [0,1]

8. Pradinė būsena:

$z(t_0) = \{0, \infty, \infty\}$

9. Perėjimo ir išėjimo operatoriai:

H(e'_{31}): / Signalas x_1 atėjo iš Siuntėjo/

if $w(e_{31}^{"}, t_m) = \infty$ then

if $((B \neq Bit3(t_m)) \wedge (P_{31} < RND(1))$ then

KSK(t_{m+1})=KSK (t_m)+1

Bit3 (t_{m+1})=B

$w(e_{31}^{"}, t_{m+1}) = t_m + \eta_{3j}$

else

$w(e_{31}^{"}, t_{m+1}) = t_m + \eta_{3,j-1}$

fi

fi

H($e_{31}^{"}$): / Patvirtinimo formavimo pabaiga/

$w(e_{31}^{"}, t_{m+1}) = \infty$

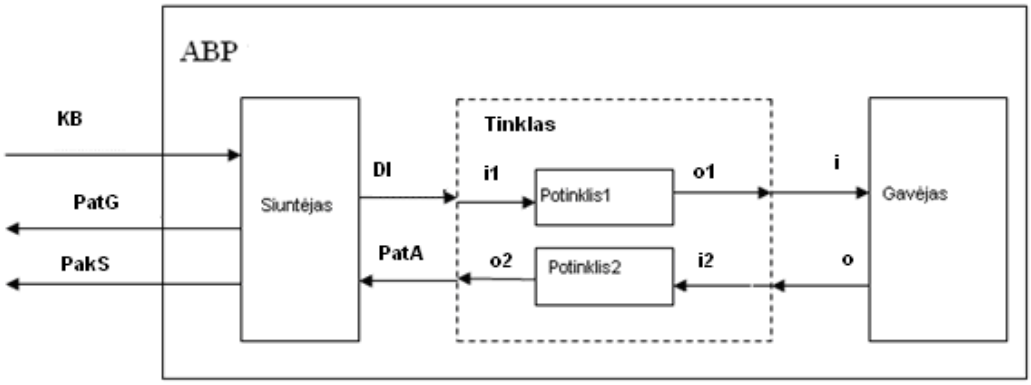
G($e_{31}^{"}$): y_1=(Bit3(t_{m+1}))

Remiantis FDEVS metodu, PLA taip pat galima sukurti naują klaidų įvedimo metodiką agregatinėje specifikacijoje, vadinamą FPLA metodu. Aprašant sistemas agregatiniu metodu idealiu atveju neatsižvelgiama į klaidas. Norint paprastam sistemos agregatiniam formalizavimui pritaikyti FPLA metodą, pirmiausia reikia formalizuoti sistemos veikimą idealiu atveju. Žinant sistemos veikimo principus, būtina išanalizuoti galimas sistemos klaidas, nuo kurių norima vėliau apsaugoti sistemą, jos veikimo metu taip, kad pati sistema įvykus klaidai galėtų su ja susidoroti. Nustačius galimas sistemos klaidas, reikia praplėsti idealų sistemos modelį, įterpiant galimas klaidas bei atitinkamai specifikuojant sistemos reakciją į įterptas klaidas. Gautasis sistemos agregatinis formalizavimas bus daug tikslesnis nei idealiu atveju, apimantis įvairius sistemos funkcionavimo atvejus, todėl modifikuota sistema bus daug patikimesnė.

5.3 FDEVS ABP specifikacija idealiu funkcionavimo atveju

5.1 ir 5.2 šio darbo skyriuose alternuojančio bito protokolas buvo nagrinėjamas idealiu atveju, kai neegzistuoja klaidos, ir kai įvertinama klaidų tikimybė, paverčiant ši protokolą klaidoms tolerantišku. Formalizuojant alternuojančio bito protokolą FDEVS metodika, taip pat atsižvelgiama į šiuos du aspektus. FDEVS metode yra naudojama ir atominis, ir jungtinis DEVS,

todėl alternuojančio bito protokolo schema yra šiek tiek modifikuojama, įsivedant daugiau parametrų. Modifikuota schema pavaizduota 10 paveiksle.



10 pav. Modifikuota alternuojančio bito protokolo schema

Kadangi norima parodyti, jog alternuojančio bito protokolo sistema gali funkcionuoti net ir klaidos atveju, įvedama papildoma [19] sąlyga – tik 90% paketų bus perduodama per potinklį. Likusieji 10% paketų yra pametami.

Gavėjas ir potinkliai turi dvi fazes (*phases*): pasyvus ir aktyvus. Šie komponentai bus fazėje „pasyvus“ pačioje pradžioje. Pasirodžius įvykiui, jie pereina iš fazės „pasyvus“ į fazę „aktyvus“ ir po tam tikro laiko tarpo išsiunčia paketą su anksčiau minėtąją 90% tikimybe. Po to grįžtama į tą pačią fazę - „pasyvus“.

Siuntėjo būsena priklauso nuo alternuojančio bito reikšmės (*B*), siuntimo režimo (*siuntimas*), patvirtinimo (*pat*) ir paketo numerio (*PN*), priklausomai nuo fazės. Siuntėjas iš pradinės fazės „pasyvus“ pereina į fazę „aktyvus“ tik tada, kai gaunamas *KB* signalas. Šis signalas priimamas tada, kai siunčiamas paketas kartu alternuojančiu bitu. Kai baigiasi siuntimo laikas, paketas laikomas išsiųstu ir siuntėjas laukia patvirtinimo apie gautą paketą iš gavėjo. Jei laukimo laikas baigiasi, taip ir neatėjus patvirtinimui, siuntėjas pakartotinai išsiunčia paketą su tuo pačiu alternuojančiu bitu. Jei patvirtinimas gautas anksčiau nei baigiasi patvirtinimo laukimui skirtas laikas, siunčiamas naujas paketas, su invertuota alternuojančio bito reikšme. Išvesties duomenys yra generuojami, kai paketas yra išsiunčiamas (*PakS*, *dt*) arba kai laukiamas patvirtinimas yra gautas (*patG*).

Skirtingai nuo sistemos formalizuotos agregatiniu metodu, formalizuojant DEVS pirmiausiai aprašomas bendras jungtinis DEVS modelis, po jo – komponentai (Siuntėjas, Potinklis, Gavėjas)

ir galiausiai – bendra formali jų specifikacija. Sistemos formaliame aprašyme naudojami kintamieji yra pavaizduoti 10 paveikslėlyje.

Pagal DEVS semantiką yra aprašoma modelio lygtis idealaus funkcionavimo (be klaidų) atveju:

$$D = [X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}, select\}]$$

Komponento Tinklas aprašymas:

$X = \{i1, i2\}$;

$Y = \{o1, o2\}$;

$D = \{potinklis1, potinklis2\}$;

$I(potinklis1) = self$;

$I(potinklis2) = self$;

$Z(potinklis1) = self$;

$Z(potinklis2) = self$;

$select(\{potinklis1, potinklis2\}) = potinklis1$.

Alternuojančio bito protokolo jungtinio modelio aprašymas:

$X = \{KP\}$ **;**

$Y = \{PS, patG\}$ **;**

$D = \{Siuntejas, Tinklas, Gavejas\}$;

$I(Siuntejas) = \{Tinklas, self\}$

$I(Tinklas) = \{Siuntejas, Gavejas\}$

$I(Gavejas) = \{Tinklas\}$

$Z(Siuntejas) = Tinklas$; $Z(Siuntejas) = self$;

$Z(Tinklas) = Siuntejas$; $Z(Tinklas) = Gavejas$;

$Z(Gavejas) = Tinklas$;

$select : (\{Siuntejas, Tinklas, Gavejas\}) = Siuntejas$;

$(\{Tinklas, Gavejas\}) = Tinklas;$

5.3.1 Komponento Siuntėjas specifikacija

$\sigma = \infty$;

$phase = Pasyvus$;

$PN = 0$; /paketo eilės numeris/

$PS = 0$; /bendras paketų skaičius/

$B = 0$; /alternuojantis bitas/

$Siuntimas = false$; /True, kai paketas siunčiamas;
False – laukiama patvirtinimo/

$pat = false$; /True – gautas lauktas patvirtinimas
False – gautas priešingas patvirtinimas/

5.3.2 Komponento Potinklis specifikacija

$S = \{Pasyvus, Aktyvus\}$;

$X = \{i\}$;

$Y = \{o\}$;

$\delta_{\text{int}}(aktyvus) = pasyvus$;

$\delta_{ext}(i, aktyvus) = aktyvus$;

$\delta_{ext}(i, pasyvus) = aktyvus$;

$\lambda(aktyvus)$

{Siųsti duomenis iš prievado *i* į prievadą *o*; (100%)
}

$ta(pasyvus) = \infty$;

$ta(aktyvus) = t_v$ /vėlinimo laikas/

5.3.3 Komponento Gavėjas specifikacija

$S = \{Pasyvus, Aktyvus\}$;

$X = \{i\}$;

$Y = \{o\}$;

$\delta_{\text{int}}(aktyvus) = pasyvus$;

$\delta_{ext}(i, aktyvus) = aktyvus$;

$\delta_{ext}(i, pasyvus) = aktyvus$;

$\lambda(aktyvus)$

{nustatyti alternuojančio bito reikšmę ir siųsti atgal

}

$ta(pasyvus) = \infty$;

$ta(aktyvus) = t_g$; /gavimo laikas/

5.3.3 Formali ABP funkcionavimo idealiomis sąlygomis specifikacija

$X = \{KB, patI\}$

$Y = \{dt, paksS, patG\}$;

$S = \{phase, \sigma, B, PN, siuntimas, pat\}$

$\delta_{ext} = \{B, PN, siuntimas, pat, e, x\}$

{case phase

Pasyvus:

if $x \in KB \cup phase$

$BPS = KB$;

$PN = 1$;

$B = PN\%2$; /siunčiamas pradinis alternuojantis bitas/

else

visi įvesties įvykiai yra ignoruojami;

Aktyvus:

if $x \in patI$

if $patI == B$ /laukiamas patvirtinimas/

{ $pat = true$;

$siuntimas = false$;

$\sigma = 0$; /iš karto suveikia vidinė perėjimų funkcija/

}

else

visi įvesties įvykiai yra ignoruojami;

}

$\delta_{\text{int}}(B, PN, siuntimas, pat, e, x)$

{case phase

Aktyvus:

If *(pat)* /gautas laukiamas patvirtinimas/

{ if *(PN<BPS)* /siųsti kitą paketą/

{ $PN = PN + 1$;

$B = (B+1)$;

$siuntimas = true$; /siunčiamas paketas/

$\sigma = t_s$;

}

else /visi paketai buvo išsiųsti sėkmingai/

{ $phase = pasyvus$; /grįžtama į pradinė pasyvią būseną/

$\sigma = \infty$;

}

}

$\lambda(aktyvus \cap siuntimas)$:

{ Siųsti paketo numerį *PN* į prievadą *pakS* /paketo eilės numeris/

Siųsti (PN*10+B) į prievadą *dt;* /žinutė su alternuojančiu bitu/

}

$\lambda(aktyvus \cap !siuntimas \cap pat)$:

{ Siųsti *B* į prievadą *PatG;* */lauktas patvirtinimas/*

}

5.4 FDEVS tolerantiško klaidoms ABP specifikacija

Šiame skyriuje (5.4) analogiškai kaip ir 5.2 skyriuje, FDEVS metodu formalizuojamas alternuojančio bito protokolas, įvertinant galimas klaidas, kurios gali įvykti protokolo funkcionavimo realiame pasaulyje metu. Jos yra aprašytos 5.3 skyriuje, bendrame alternuojančio bito protokolo sistemos funkcionalumo aprašyme.

Bendras jungtinis DEVS modelis:

$D = [X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}, select\}]$

Komponento Tinklas aprašymas:

$X = \{i1, i2\}$;

$Y = \{o1, o2\}$;

$D = \{potinklis1, potinklis2\}$;

$I(potinklis1) = self$;

$I(potinklis2) = self$;

$Z(potinklis1) = self$;

$Z(potinklis2) = self$;

$select(\{potinklis1, potinklis2\}) = potinklis1$.

Alternuojančio bito jungtinio modelio aprašymas:

$X = \{KP\}$ **;**

$Y = \{PS, patG\}$ **;**

$D = \{Siuntejas, Tinklas, Gavejas\}$;

$I(Siuntejas) = \{Tinklas, self\}$

$I(Tinklas) = \{Siuntejas, Gavejas\}$

$I(Gavejas) = \{Tinklas\}$

$Z(Siuntejas) = Tinklas$; $Z(Siuntejas) = self$;

$Z(Tinklas) = Siuntejas$; $Z(Tinklas) = Gavejas$;

$Z(Gavejas) = Tinklas$;

$select$: $(\{Siuntejas, Tinklas, Gavejas\}) = Siuntejas$;

$(\{Tinklas, Gavejas\}) = Tinklas$;

5.4.1 Komponento Siuntėjas specifikacija

$\sigma = \infty$;

$phase = Pasyvus$;

$PN = 0$; /paketo eilės numeris/

$PS = 0$; /bendras paketų skaičius/

$B = 0$; /alternuojantis bitas/

$Siuntimas = false$; /True, kai paketas siunčiamas;
False – laukiama patvirtinimo/

$pat = false$; /True – gautas lauktas patvirtinimas

False – gautas priešingas patvirtinimas/

5.4.2 Komponento Potinklis specifikacija

$S = \{Pasyvus, Aktyvus\}$;

$X = \{i\}$;

$Y = \{o\}$;

$\delta_{\text{int}}(aktyvus) = pasyvus$;

$\delta_{ext}(i, aktyvus) = aktyvus$;

$\delta_{ext}(i, pasyvus) = aktyvus$;

$\lambda(aktyvus)$

{Siųsti duomenis iš prievado *i* į prievadą *o*; (90%)
Laikyti, kad paketai yra pamesti (10%)
}

$ta(pasyvus) = \infty$;

$ta(aktyvus) = t_v$ /vėlinimo laikas/

5.4.3 Komponento Gavėjas specifikacija

$S = \{Pasyvus, Aktyvus\}$;

$X = \{i\}$;

$Y = \{o\}$;

$\delta_{\text{int}}(aktyvus) = pasyvus$;

$\delta_{ext}(i, aktyvus) = aktyvus$;

$\delta_{ext}(i, pasyvus) = aktyvus$;

$\lambda(aktyvus)$

{nustatyti alternuojančio bito reikšmę ir siųsti atgal
}

$ta(pasyvus) = \infty$;

$ta(aktyvus) = t_g$; /gavimo laikas/

5.4.4 Formali tolerantiško klaidoms ABP funkcionavimo specifikacija

$X = \{KB, patI\}$

$Y = \{dt, paksS, patG\}$;

$S = \{phase, \sigma, B, PN, siuntimas, pat\}$

$\delta_{ext} = \{B, PN, siuntimas, pat, e, x\}$

{case phase

Pasyvus:

if $x \in KB \cup phase$

$BPS = KB$;

$PN = 1$;

$B = PN\%2$; /siunčiamas pradinis alternuojantis bitas/

$pat = false$; /negautas laukiamas patvirtinimas/

$siuntimas = true$; /paketas siunčiamas/

$\sigma = t_s$; /pereinama į siunčiamo paketo būseną/

$phase = active$;

else

visi įvesties įvykiai yra ignoruojami;

Aktyvus:

if $x \in patI$

if $patI == B$ /laukiamas patvirtinimas/

{ $pat = true$;

$siuntimas = false$;

$\sigma = 0$; /iš karto suveikia vidinė perėjimų funkcija/

}

else

visi įvesties įvykiai yra ignoruojami;

}

$\delta_{\text{int}}(B, PN, siuntimas, pat, e, x)$

{case phase

Aktyvus:

If *(pat)* /gautas laukiamas patvirtinimas/

{ if *(PN<BPS)* /siųsti kitą paketą/

{ $PN = PN + 1$;

$B = (B+1)$;

$siuntimas = true$; /siunčiamas paketas/

$pat = false$; /neatėjo lauktas patvirtinimas/

$\sigma = t_s$;

}

else /visi paketai buvo išsiųsti sėkmingai/

{ $phase = pasyvus$; /grįžtama į pradinė pasyvią būseną/

$\sigma = \infty$;

}

}

else if (*siuntimas*) /pereiti į laukimo būseną po paketo siuntimo/

{ $siuntimas = false$; /laukiama patvirtinimo/

$\sigma = t_o$; /baigėsi laukimui skirtas laikas/

}

else /laukimo laikas baigėsi, persiųsti prieš tai buvusį paketą/

{ $siuntimas = true$; /paketas siunčiamas/

$\sigma = t_s$; /siuntimo laikas/

}

$\lambda(aktyvus \cap siuntimas)$:

{ Siųsti paketo numerį *PN* į prievadą *pakS* /paketo eilės numeris/

Siųsti (PN*10+B) į prievadą *dt;* /žinutė su alternuojančiu bitu/

}

$\lambda(aktyvus \cap !siuntimas \cap pat)$:

{ Siųsti *B* į prievadą *PatG;* */lauktas patvirtinimas/*

}

}

DARBO REZULTATAI IR IŠVADOS

Šiame darbe atlikto tyrimo metu buvo sukurta FPLA modeliavimo metodika, iliustruota modifikuoto alternuojančio bito agregatine specifikacija, nagrinėjant ją klaidoms tolerantiškos sistemos požiūriu.

Parodyta, kad formalizuojant klaidoms tolerantiškas sistemas, aprašytas PLA metodu, būtina atlikti šią analizę bei veiksmus nustatyta tvarka:

1. Išanalizuoti sistemos veikimą idealiu jos funkcionavimo atveju, darant prielaidą, kad bet kuris sistemos vykdomas veiksmas bus atliekamas teisingai, be jokių sutrikimų.
2. Apsibrėžti galimų klaidų aibę, kurios gali įvykti ateityje, sistemos funkcionavimo realiame laike metu.
3. Žinant galimas sistemos klaidas, praplėsti formalų sistemos PLA aprašymą, skirtą jos funkcionavimui idealiomis sąlygomis specifikuoti, įvedant klaidas ir formalizuojant, kaip sistema turės į jas reaguoti.

Ateityje šią metodiką galima plėtoti toliau verifikuojant ir validuojant sukurtas formalias specifikacijas su tikslu parodyti, kad klaidų aibės nustatymas ir jų įvedimas į formalią sistemos specifikaciją leidžia sistemai jos veikimo metu realiame laike teikti paslaugas net ir klaidos egzistavimo atveju.

LITERATŪRA

1. ZHAO, L; LIZHENG, F; JIANPING, W. A New Method of Fault Tolerance TCP, Department of Computer Science, Tsinghua University, 2003. [Žiūrėta 2006-12-11]. Prieiga internete - http://ieeexplore.ieee.org/iel5/8807/27858/01243062.pdf
2. HEIMERDINGER, W.;WEINSTOCK, C. *A Conceptual Framework for System Fault Tolerance*. 1992.
3. RANDELL, B; XU, J. *The Evolution of the Recovery Block Concept*. University of Newcastle upon Tyne, 2002. [Žiūrėta 2006-12-11]. Prieiga internete - http://www.cs.ncl.ac.uk/research/pubs/books/papers/101.pdf.
4. VOAS, J; MCGRAW, G; KASSAB, L. A „Crystal Ball" for Software Liability, 1997. [Žiūrėta 2007-01-13]. Prieiga internete - http://ieeexplore.ieee.org/iel1/2/12848/00587545.pdf?arnumber=587545
5. AVIZIENIS, A. *Toward Systematic Design of Fault-Tolerant Systems*, 1997. [Žiūrėta 2007-04-11]. Prieiga internete - [Žiūrėta 2006-12-11]. Prieiga internete http://ieeexplore.ieee.org/iel1/2/12687/00585154.pdf?arnumber=585154
6. *FDEVS: a general devs-based formalism for fault modeling and simulation* [žiūrėta 2006-02-01], prieiga internete - http://www.fceia.unr.edu.ar/~kofman/files.
7. PRANEVIČIUS, H. *Kompiuterių tinklų protokolų formalusis specifikavimas ir analizė: agregatinis metodas*. Kaunas, 2005. 80-85p.
8. ZEIGLER, B.P.; PRAEHOFER, H.; KIM, T.G. *Theory of Modeling and Simulation Second Edition. Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press, 2000, 7-87 p. ISBN-13: 978-0-12-778455-7.
9. JOON SUNG HONG, HAE SANG SONG, TAG GON KIM. *A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development.* 1997.
10. ZEIGLER, B.; SUNGDO, CHI. *Symbolic Discrete Event System Specification.* 2002.
11. RUDIE, K.; WONHAM, M. *Protocol Verification Using Discrete-Event Systems.* [žiūrėta 2006-02-01], prieiga internete - http://citeseer.ist.psu.edu/cache/papers/cs/7067/http:zSzzSzwww.ima.umn.eduzSzpreprintszSzNOVEMBER1992zSz1043.pdf/rudie92protocol.pdf.
12. FUYAU, L. A *Formalism for Specifying Communicating Processe.* [žiūrėta 2006-02-25], prieiga internete - http://portal.acm.org/citation.cfm?id=170791.170816.

13. HAMMARBERG J.; NADJM-TEHRANI S. *Formal verification of fault tolerance in safety-critical reconfigurable modules.* 2004.
14. VANGHELUWE H. *The Discrete Event System Specification (DEVS) formalism*. 2002..
15. ARLAT J.; CROUZET Y.; LAPRIE J.C. *Fault injection for dependability validation of fault – tolerant computing systems*. 2000.
16. AVIŽIENIS, A.; LAPRIE J.C.; RANDELL B. *Fundamental Concepts of Dependaility*. 2000.
17. STEVENSON, D.E. *From DEVS to formal methods: a categorical approach*. 1995.
18. ADEMAJ, A.; HLAVICKA, J. *Fault Tolerance Evaluation Using Two Software Based Fault Injection Methods*. 2002
19. ZHEN, T. *Modelling Discrete-Event Systems Using DEVS*. 2004 [žiūrėta 2006-02-15], prieiga internete - http://www.informs-sim.org/wsc06papers/103.pdf.
20. LITLLEWOOD, B.; STRIGINI, L. *Software Reliability and Dependability: a Roadmap.* 2006
21. HSUEH, M.; TSAI, T.; IYER, R. Fault *Injection Techniques and Tools*. 1997 [Žiūrėta 2007-03-08] Prieiga internete - http://dslab.epfl.ch/courses/pods/winter06-07/readings/hsueh-injection.pdf.
22. ZEIGLER; B. DEVS representation of Dynamical Systems: Event Based Intelligent Control, 1999. [Žiūrėta 2006-12-14]. Prieiga internete - http://ieeexplore.ieee.org/iel5/5/859/00021071.pdf?tp=&isnumber=&arnumber=21071

„Systems With Possibility To Insert Faults Formalization“

SANTRAUKA ANGLŲ KALBA (SUMMARY)

Currently many very high requirements are raised for the real time systems. These systems must be of high availability, should have a very high dependability and should be safe. Nowadays quite an important goal is to create a fault tolerant system. Such system should be enabled to work in a case of the appeared fault. The system should not fail and should continue rendering defined services and generating proper output results.

In order to create a fault tolerant system, very clear requirements should be prepared and all possible fault events should be analyzed. It can be properly made by using any of system modeling formalism. In this work alternating bit protocol system was chosen to formalize and analyzed in fault tolerant software aspects. Alternating bit protocol was modified in two ways – it's functionality under perfect circumstances and with added faults, in order to make the system fault tolerant. These both cases were formalized by PLA and DEVS formalization methods. After the research of different formalisms and adjusting FDEVS to alternating bit protocol, FPLA formalization method was created.